| | |
|---|---|
| **Titre:** Title: | Conception et implémentation d'un treillis de calcul configurable à deux niveaux |
| **Auteur:** Author: | Mathieu Allard |
| **Date:** | 2012 |
| **Type:** | Mémoire ou thèse / Dissertation or Thesis |
| **Référence:** Citation: | Allard, M. (2012). Conception et implémentation d'un treillis de calcul configurable à deux niveaux [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1004/ |

UNIVERSITÉ DE MONTRÉAL

# CONCEPTION ET IMPLÉMENTATION D'UN TREILLIS DE CALCUL CONFIGURABLE À DEUX NIVEAUX

MATHIEU ALLARD
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 2012

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

CONCEPTION ET IMPLÉMENTATION D'UN TREILLIS DE CALCUL CONFIGURABLE À DEUX NIVEAUX

présenté par : ALLARD Mathieu
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury d'examen constitué de :

M. SAVARIA Yvon, Ph.D., président
M. DAVID Jean Pierre, Ph.D., membre et directeur de recherche
M. BOYER François-Raymond, Ph.D., membre

*Always pass on what you have learned.*
*-Yoda*

# REMERCIEMENTS

J'aimerais profiter de cette occasion pour remercier mon directeur de recherche, le professeur Jean Pierre David. Au travers de nos discussions, il m'a apporté une compréhension plus approfondie des divers aspects du sujet. Je salue également sa grande ouverture d'esprit et son professionnalisme qui m'ont appris que toutes les avenues sont possibles si la rigueur et la persévérance font partie du travail, ce que je préconise par dessus tout. Il a su, dès la fin de mon baccalauréat, me donner la piqûre pour le domaine des architectures sur FPGA de par son enseignement et ses connaissances. Je le remercie également de m'avoir permis de donner des charges de laboratoire qui m'ont fait découvrir une nouvelle passion. Mes multiples travaux avec lui au cours de ma maîtrise sont pour moi des expériences inestimables pour le fondement de ma carrière professionnelle.

Je souhaite également remercier le professeur Yvon Savaria qui, bien qu'il ne fut pas mon co-directeur, apporta de judicieux conseils lors de nos nombreuses rencontres. Que ce soit pour ses recommandations au niveau de la rédaction d'articles scientifiques ou pour ses connaissances dans le domaine de la microélectronique, ses suggestions furent toujours prises avec beaucoup d'importance et de soins. J'aimerais également remercier François Gagnon qui fut mon contact chez le partenaire industriel et qui me donna la possibilité de poursuivre le projet à travers ces années tout en me prodiguant de bonnes explications sur les différents enjeux du design. Je remercie aussi MITACS et le partenaire industriel pour la bourse offerte à mon égard, ce qui m'a permis de poursuivre mes études.

En terminant, j'aimerais saluer et remercier mes collègues de laboratoire Tarek, Marc-André, Patrick, Jonas et Adrien. Nos nombreuses discussions, que ce soit sur des sujets scientifiques ou non, laissent en moi le souvenir de notre profonde amitié développée au cours du temps. J'espère avoir la chance et l'honneur de retravailler avec vous dans le futur. Je remercie également ma famille qui a toujours été là pour me supporter et m'encourager inconditionnellement à travers mes études. La dernière personne, et non la moindre, que j'aimerais remercier est ma conjointe Karine. Les mots me manquent pour te remercier, à ta juste valeur, pour le soutien moral et psychologique indispensable que tu as fourni au travers des aléas de la vie. Ta patience, ta compréhension et ton écoute m'ont sincèrement permis d'amener ce projet à terme. Merci pour tout.

# RÉSUMÉ

De nos jours, la technologie FPGA est devenue de plus en plus puissante et complexe à un niveau que seule la technologie ASIC pouvait atteindre il y a quelques années. Les FPGA peuvent inclure plusieurs processeurs, des unités de traitement spécialisées, des réseaux sur puce pour le routage des données en interne, etc. Bien que le FPGA fonctionne à une fréquence moindre qu'un processeur à usage général, la nature parallèle de la logique matérielle permet tout de même d'opérer des algorithmes beaucoup plus rapidement. En combinant les meilleures propositions d'un système basé sur processeur ou d'un ASIC, les FPGA ont ouvert une grande flexibilité et la possibilité d'un prototypage rapide pour les ingénieurs et scientifiques de toutes les expertises.

Toutefois, malgré les nombreux progrès réalisés dans la description de systèmes matériels à des niveaux d'abstraction plus élevés (des modules IP configurables, des réseaux sur puces, des processeurs configurables etc.), la réalisation d'architectures complexes est encore aujourd'hui un travail réservé aux spécialistes (typiquement des ingénieurs en conception de circuits numériques). De plus, c'est un processus relativement long (des mois, voire des années) si on considère non seulement le temps de conception de l'architecture mais encore le temps pour la vérifier et l'optimiser. De plus, les ressources disponibles augmentent à chaque nouvelle mouture, menant à des architectures de plus en plus élaborées et à une gestion de plus en plus difficile. Le FPGA proposerait alors des performances et un pouvoir de calcul inégalés, mais serait difficilement utilisable car on n'arriverait pas à extraire de façon simple et efficace toute cette puissance.

Le but ultime serait d'avoir des performances du niveau matériel avec la flexibilité et la simplicité de développement du logiciel. Le projet consiste à faire la conception et l'implémentation d'une toute nouvelle architecture de type treillis permettant de traiter des algorithmes sur un grand flot de données. Les algorithmes ayant de grandes possibilités de parallélisme seraient avantagés par ce treillis. Ce treillis de calcul est configurable à deux niveaux d'abstraction. Au plus bas niveau (niveau matériel), l'architecture est constituée de divers blocs permettant de réaliser les différents chemins de données et de contrôle. Les données se propagent donc de mémoires en mémoires à travers des ALU. Ces transactions sont contrôlées par le niveau plus haut (niveau de configuration logicielle). En effet, l'utilisateur du treillis pourra venir implémenter ses algorithmes à travers de petites mémoires à instructions situées dans l'architecture. Il sera également possible de venir reconfigurer dynamiquement le

comportement du treillis. On permet donc à des programmeurs logiciels d'exploiter toute la puissance d'une implémentation matérielle, sans toutefois devoir la développer en détail. Cela évite aux utilisateurs d'avoir à apprendre à exploiter les FPGA à un niveau matériel, tout en gardant confidentielle le détail interne de la dite architecture.

Ce travail s'inscrit dans le cadre d'un partenariat industriel avec une société finançant le projet en collaboration avec l'organisation MITACS. Cette société fabrique et commercialise des cartes d'acquisition opérant à haute fréquence et offrant de hautes résolutions. Ces cartes utilisent déjà des FPGA dont une partie importante de la logique est présentement inexploitée. Les clients demandent de pouvoir utiliser cette logique pour réaliser un prétraitement de l'information. Toutefois, la plupart d'entre eux seraient incapables de l'utiliser à bon escient, c'est-à-dire de manière efficace et sans nuire au reste de la logique nécessaire au bon fonctionnement de la carte. De plus, le partenaire ne désire pas divulguer le code source de l'architecture implantée dans le FPGA. La pertinence du projet proposé se justifie donc par le fait qu'il serait possible de livrer des cartes avec la dite architecture (en plus de la logique déjà utilisée). Le client spécifierait seulement ses applications à haut-niveau.

Ce document démontre comment réaliser une telle architecture. Elle se compose de 3 modules distincts formant les tuiles du tissu qui sont finement construites de sorte à permettre la plus grande flexibilité possible pour une complexité moindre. Les *Machines à états* contiennent les mémoires d'instructions du système. Elles représentent l'accès haut-niveau qui donne son sens au treillis. Elles communiquent avec des *SALU* à travers un réseau de *Banques de FIFO* qui emmagasinent les différents jetons de donnée, permettant ainsi de créer le flot de données. Les *SALU* proprement dites servent à faire tous les calculs nécessaires. Le pipeline et le chemin de données sont des éléments essentiels au bon fonctionnement des algorithmes implémentés.

Les résultats de l'implémentation sont très encourageants. Au niveau des ressources, l'implémentation sur un Stratix III EP3SL150F1152C2N donne qu'une *Banque de FIFO* utilise 13% des ALM disponibles, alors qu'une *SALU* en utilise 15%. Pris comme tels, ces chiffres sont de bonnes tailles et peuvent paraître comme un énorme inconvénient à utiliser cette architecture. Cependant, un usager qui ne désire pas avoir l'option de la reconfiguration dynamique pourra la désactiver et obtenir des pourcentages d'utilisation plus faibles. De plus, si on considère un FPGA plus récent, soit le Sratix IV GX530, les taux ci-haut s'abaissent à 3% et 4%. Pour un Stratix V GXBB, on obtient 1.8% et 2.4%. Plusieurs algorithmes furent implémentés sur le treillis, dont un filtre FIR. Au final, nous obtenons un débit d'une sortie

par coup d'horloge à une fréquence environnant les 200MHz. Finalement, la reconfiguration dynamique est possible sur le treillis et elle permet ainsi de venir changer le comportement du flot de données sans avoir à reprogrammer le FPGA, offrant ainsi un énorme avantage sur ses compétiteurs. Bien entendu, le but n'était pas d'utiliser les ressources sans jugement et que selon leur disponibilité, mais également d'avoir une architecture optimisée et performante, ayant un caractère générique et démontrant une facilité de mise en oeuvre.

# ABSTRACT

Nowadays, FPGA technology has become more powerful and complex at a level that only ASIC could reach a few years ago. FPGAs can now include several processors, specialized processing units, on-chip networks for routing internal data, etc. Although the FPGA operates at a smaller frequency than a general purpose processor, the parallel nature of the hardware logic still allows algorithms to operate much faster. By combining the best parts of a system based on processor or a ASIC, FPGAs have allowed a great deal of flexibility and the possibility of rapid prototyping for engineers and scientists of all expertise.

However, despite the many advances in hardware systems described at higher levels of abstraction, the implementation of complex architecture is still a job for specialists (electronic engineers designing digital circuits). Moreover, it is a relatively long process (months or years) if we consider not only the time to design the architecture but also the time to verify and optimize it. In addition, available resources are increasing in each new FPGA version, leading architectures to become more elaborate and management more difficult. The FPGA proposes very high performance and unrivaled computing power, but may become obsolete because it is difficult to extract in a simple and effective way all that power.

The ultimate goal would be to have hardware performance with software development simplicity and flexibility. The project involves the design and implementation of a new architecture (mesh type) for processing algorithms on a large data stream. Algorithms with great potential for parallelism would benefit from this lattice. The mesh is configurable at two levels of abstraction. At the lowest level (hardware level), the architecture consists of various blocks supporting the different data paths and control structures. The data thus propagate from memories to memories through ALUs. These transactions are controlled by a higher representation (software level). In fact, the user can implement algorithms on the lattice through small memories placed within the architecture. With the proposed structure, it is also possible to dynamically reconfigure the behavior of the lattice. It thus allows software programmers to harness the power of a hardware implementation, without further notice. This prevents users from having to learn how to use the FPGA while allowing to keep confidential the architecture itself.

This work is part of an industrial partnership with a company financing the project with a MITACS internship. The company manufactures and markets frame grabbers operating at

high frequencies and offering high resolutions. These cards are already using FPGAs with an important part of these circuits being unused logic. Customers are asking to use this logic to perform preprocessing algorithms. However, most of them would be unable to use it effectively and without harming the rest of the logic required for proper operation of the board. In addition, the partner does not want to disclose the source code of the architecture implemented in the FPGA. The relevance of the proposed project is therefore justified by the fact that it would be possible to deliver boards with the architecture (in addition to the logic already used). The customer would only specify his high-level applications.

This work demonstrates the implementation of this architecture. Three separate modules forming the fabric tiles are constructed to allow the greatest possible flexibility at a lesser complexity. The *States Machine* contains the instruction memories of the system. They constitute the high-level access to the mesh. They communicate with $SALU$ through a network of *FIFO Banks* that store different data tokens, thereby creating the data stream. The $SALU$ are used to make all the necessary calculations. The pipeline and the data path are essential for the proper functioning of the implemented algorithms.

The reported implementation results are very encouraging. When looking at resources, the implementation on a Stratix III EP3SL150F1152C2N gives *FIFO Banks* a 13% usage of available ALMs, and 15% for a $SALU$. Taken as such, these figures may seem large and appear to be a huge drawback for using this architecture. However, a user who does not wish to have the option of dynamic reconfiguration can disable it and get utilization reduced. Moreover, if we consider a newer FPGA, namely the Sratix IV GX530, the above rates are reduced to 3% and 4%. For a Stratix V GXBB, we obtain 1.8% and 2.4%. Several algorithms were implemented on the mesh, including a FIR filter. In the end, we get a rate of one output per clock cycle at a frequency approaching 200MHz. Finally, dynamic reconfiguration is possible on the lattice and allows for behavior changes of the data stream without having to reprogram the FPGA, providing a significant advantage over its competitors. Of course, the goal was not to use the resources without judgment and according to their availability, but also to have an optimized architecture and with great performance and demonstrating an ease of implementation.

# TABLE DES MATIÈRES

# LISTE DES TABLEAUX

# LISTE DES FIGURES

# LISTE DES SIGLES ET ABRÉVIATIONS

| | |
|---|---|
| ALM | Adaptive Logic Modules |
| ALU | Arithmetic Logic Unit |
| ASIC | Application-Specific Integrated Circuit |
| CASM | Channel Based Algorithmic State Machine |
| CGRA | Coarse-Grained Reconfigurable Architecture |
| CPLD | Complex Programmable Logic Device |
| DDC | Digital Down Converter |
| DSP | Digital Signal Processor |
| FIFO | First In, First Out |
| FIR | Finite Impulse Response |
| FPGA | Field Programmable Gate Array |
| FPOA | Field Programmable Object Array |
| GAL | Generic Array Logic |
| I/O | Input/Output |
| IP | Intellectual Property |
| MIMD | Multiple Instruction, Multiple Data |
| MISD | Multiple Instruction, Single Data |
| NoC | Network on Chip |
| PAL | Programmable Array Logic |
| PLA | Programmable Logic Array |
| PLD | Programmable Logic Device |
| PROM | Programmable Read-Only Memory |
| RISC | Reduced Instruction Set Computing |
| RTR | Ready To Receive |
| RTS | Ready To Send |
| SALU | Shared Arithmetic Logic Unit |
| SIMD | Single Instruction, Multiple Data |
| SISD | Single Instruction, Single Data |
| SoC | System on Chip |
| UF | Unité Fonctionnelle |
| VHDL | Very-high-speed-integrated-circuits Hardware Description Language |

# CHAPITRE 1

# INTRODUCTION

De nos jours, l'électronique est devenue tellement omniprésente qu'il devient aisé d'oublier son importance dans notre quotidien. En effet, des milliers de dispositifs rendent nos vies plus faciles, agrémentent notre quotidien, nous permettent d'aller travailler et de profiter de nos loisirs sans vraiment considérer à quel point tout cela est devenu pratique. C'est un réveille-matin qui nous sort du lit pour que l'on puisse déjeuner à l'aide du four à micro-ondes ou du grille-pain. Nous utilisons ensuite la voiture pour vaquer à nos occupations journalières alors que plus de la moitié de son coût de fabrication est associé à l'électronique qu'elle contient. On utilise ensuite notre ordinateur personnel, on accède à des serveurs internet situés à l'autre bout de la planète, on écoute la télévision, on prend l'ascenseur, on écoute de la musique sur un lecteur MP3, on prend une collation de la machine distributrice, on parle à un ami par l'entremise d'un téléphone cellulaire, etc. Notre style de vie actuel ne pourrait se poursuivre sans l'utilisation des avancées dans le domaine de l'électronique.

Le concept va cependant beaucoup plus loin. Les exemples cités précédemment affectent directement la vie des gens qui les utilisent, mais il existe des applications de l'électronique que le commun des mortels ne soupçonne pas. Par exemple, lorsque deux personnes utilisent le téléphone pour communiquer, on ne se rend pas compte qu'il y a tout un réseau établi pour permettre la connexion. Le domaine de la télécommunication regorge d'innovations (*VoIP-Voice over IP*, instruments de test et de mesure, satellites, etc.) qui nécessitent le travail de plusieurs ingénieurs experts en la matière. En aérospatiale, le concept de robustesse devient inestimable car les systèmes sont extrêmement complexes et les risques très importants. Toutes les notions d'électronique analogique et numérique sont utilisées afin de parvenir aux objectifs finaux. Ces enjeux sont transparents pour la plupart des gens mais ils nécessitent des progrès constants.

Un des modèles de référence pour puiser de nouvelles idées est la nature. Cette mécanique bien huilée propose de nombreux cas qui se traduisent au niveau de l'électronique. À titre d'exemple, plusieurs recherches tentent de recréer le concept d'anticorps afin d'envoyer des nanorobots à l'intérieur du corps humain pour guérir les cancers. Également, en comprenant le fonctionnement du cerveau, on pourrait reproduire certains concepts sur les ordinateurs et leur façon d'accéder aux ressources. Un thème qui revient régulièrement et qu'on retrouve

partout est le parallélisme. Le monde dans lequel on vit est de nature fondamentalement parallèle et cette application au niveau des systèmes électroniques apporte des résultats très impressionnants que nous étudierons prochainement.

L'avènement des premiers microprocesseurs dans les années 1970 déclencha la roue du progrès dans le domaine de l'électronique numérique. Provenant de l'évolution des transistors et du circuit intégré, ils sont la source de l'existence des appareils électroniques que nous utilisons à chaque jour, tels que les ordinateurs, les cellulaires et même les voitures. D'ailleurs, il n'est pas faux d'affirmer, en regardant l'histoire, que c'est cette technologie qui repousse les limites du nombre de transistors présents sur une seule puce. La loi de Moore, qui stipule que cette quantité double approximativement à tous les deux ans, s'inscrit parfaitement dans cette pensée. Néanmoins, un aspect important pousse rapidement les chercheurs à trouver de nouvelles solutions : le souci de performance. Les principales motivations sont la hausse de la fréquence d'utilisation, le meilleur débit possible ainsi qu'un immense pouvoir de calcul. L'idée de pouvoir contrôler le chemin de données et le chemin de contrôle devint alors de plus en plus intéressante. La taxonomie de Michael J. Flynn, que l'on retrouve en figure 1.1, permet de classer les systèmes informatiques selon ces notions (les UF ou Unités Fonction-

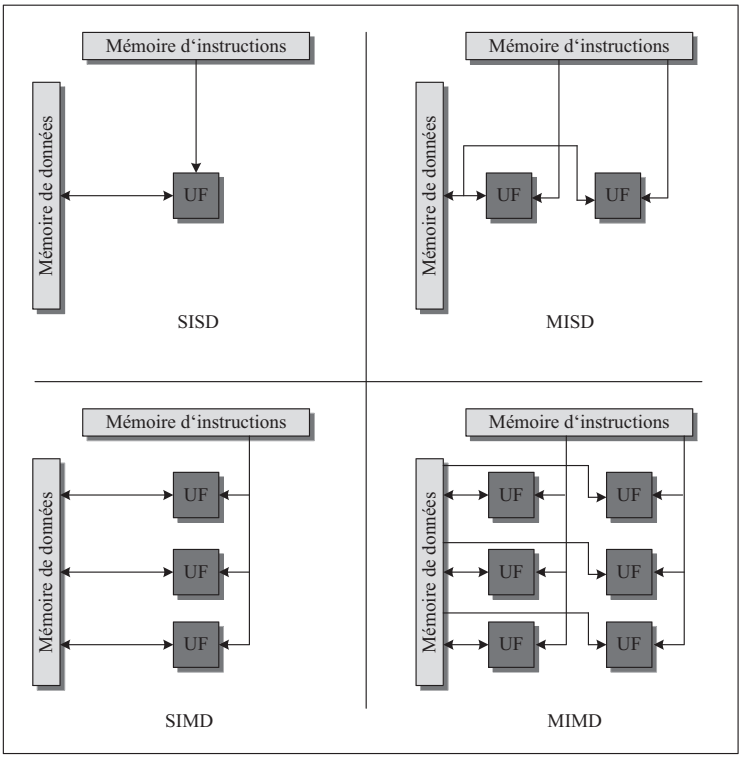


Figure 1.1 Classification de Flynn des systèmes informatiques

nelles peuvent représenter autant des processeurs que du matériel dédié). Le SISD (*Single Instruction, Single Data*) est le plus simple et correspond à l'architecture Harvard, soit une seule instruction par emplacement dans la mémoire de données. Les classes MISD (*Multiple Instruction, Single Data*), SIMD (*Single Instruction, Multiple Data*) et MIMD (*Multiple Instruction, Multiple Data*) sont des dérivations et essaient de profiter le plus possible de la notion de parallélisme. En prenant le MIMD présent sur la figure 1.1, on pourrait théoriquement effectuer 2 instructions différentes, chacune sur un lot de 3 Unités Fonctionnelles pour 6 jeux différents de données sur le même cycle. Au niveau d'un SISD, cela prendrait 6 cycles d'opérations. Un processeur multi-coeur, par exemple, est une machine MIMD. Bien que cette classification date de 1966, elle est toujours actuelle et montre parfaitement tous les intérêts du parallélisme.

Donc, dans la quête continue de consommation de puissance et du coût le plus faible tout en observant les meilleures performances possibles, le calcul parallèle devint inévitable. Le parallélisme vient du principe qu'un problème peut se diviser en de plus petites parties qui peuvent être traitées de façon concurrentielle. Théoriquement, on peut alors grandement accélérer une application. Néanmoins, les procédés concurrentiels sont plus difficiles à définir qu'un simple programme séquentiel appliqué par un processeur. La communication et la synchronisation peuvent devenir des obstacles qui freinent radicalement les performances escomptées. Le résultat de l'implémentation parallèle d'une fonction peut être mis en relation avec la loi de Amdahl qui caractérise l'accélération totale de la façon suivante :

$$\frac{P}{1 + f * (P - 1)}$$

où P est l'accélération de la tâche en question et f la fraction du temps pris par toutes les autres fonctionnalités. En d'autres termes, il ne suffit pas tout simplement de paralléliser n'importe quelle partie d'un algorithme pour en retirer un bénéfice important. Une architecture offrant la possibilité de faire et de gérer des calculs parallèles permettrait d'offrir des performances sans précédent et d'ouvrir la porte aux débits de plus en plus imposants.

En 1956, Wen Tsing Show inventa l'ancêtre des architectures reconfigurables actuelles : la PROM (*Programmable Read-Only Memory*). Cette innovation de l'époque provenait d'une demande de *United States Air Force* d'avoir une manière plus flexible et sécuritaire de sauvegarder leurs données. Bien que cette mémoire ne fut pas reprogrammable, il était possible de la programmer une seule fois après sa fabrication, ce qui était révolutionnaire pour son temps. Cette notion d'un état vierge lors de la manufacture fut grandement reprise dans le

concept des PLD (*Programmable Logic Device*). Ce sont des composants électroniques qui permettent de décrire un circuit numérique et sont reprogrammables. Avec le temps, la possibilité de reconfigurer les PROM par ultraviolet ou de façon électrique devint accessible, mais plusieurs inconvénients nuisaient à leur succès :

- Ils sont généralement beaucoup plus lents que de la logique dédiée ;
- Ils consomment plus de puissance ;
- Ils coûtent plus cher que la logique programmable, spécialement lorsqu'une grande vitesse est requise ;
- Les sorties peuvent occasionnées des *glitchs* pour certaines transitions asynchrones aux entrées.

Les premiers PLD attaquèrent le marché au début des années 1970 par des compagnies telles que Motorola, Texas Instruments, General Electric et National Semiconductor. Les premiers modèles étaient des PLA (*Programmable Logic Array*) et consistaient en une porte ET à plusieurs entrées qui se connecte à une porte OU pour former une sortie représentant une somme de multiplications binaires. Par la suite, les PAL (*Programmable Array Logic*) innovèrent le concept en éliminant le champs de connexions vers la porte OU, ce qui rendit la puce plus petite, plus rapide et moins dispendieuse. Les GAL (*Generic Array Logic*) apparurent en 1985 et offraient les mêmes propriétés que ses prédécesseurs, mais autorisaient la reprogrammation. Les CPLD (*Complex Programmable Logic Device*) vinrent ensuite fournir beaucoup plus de ressources disponibles aux concepteurs. Or, malgré le succès commercial de ces systèmes, une branche de développement séparée apporta une nouvelle technologie : les FPGA (*Field-Programmable Gate Array*).

Un FPGA est un circuit qui, dans la même veine que ses compétiteurs, permet à l'utilisateur de venir programmer des fonctions logiques au niveau matériel. La différence réside dans le fait que cette puce est constituée de plusieurs petites tables de vérité qui recréent le comportement du circuit, à l'instar d'une mer de portes formant une somme de produits. Xilinx commercialisa le tout premier prototype viable en 1985 sous le nom XC2064. Il contenait 64 blocs logiques configurables avec 2 tables de vérité à 3 entrées. Le marché des FPGA en 1987 était de 14 millions de dollars et a atteint aujourd'hui près des 3 milliards. Initialement utilisé dans le domaine des télécommunications et du réseautage, on le retrouve maintenant dans presque tous les champs d'expertise en microélectronique. Bien que le FPGA fonctionne à une fréquence moindre qu'un processeur, la nature parallèle de la logique matérielle permet tout de même d'opérer des algorithmes beaucoup plus rapidement.

La comparaison entre les FPGA et les ASIC(*Application-Specific Integrated Circuit*) est de mise pour bien comprendre nos recherches. En premier lieu, un ASIC est un circuit intégré spécialisé, au contraire d'une architecture générale. Historiquement, les architectures sur FPGA étaient beaucoup plus lentes (environ 3 fois), consommaient plus d'énergie (environ 12 fois) et possédaient moins de fonctionnalités (environ 40 fois) que leur pendant sur ASIC. Cependant, avec le temps, les progrès réalisés par les grandes compagnies telles que Xilinx et Altera améliorèrent ces heuristiques. De nos jours, la technologie FPGA est devenue une solide alternative au ASIC pour des implémentations de systèmes complexes. Ces systèmes peuvent inclure, par exemple, plusieurs processeurs, des ALU et de la logique dédiée, le tout relié par un réseau de communication formant un NoC (*Network-On-Chip*). Comme nous l'avons vu précédemment, l'exploitation du parallélisme au niveau matériel permet de surpasser la puissance de calcul d'un DSP. De plus, tant que le FPGA n'est pas saturé, l'ajout de blocs de calcul ne vient pas nuire aux performances car il n'y a pas de compétition pour accéder aux ressources. Un atout majeur face à un ASIC, outre la reconfiguration, est la rapidité de mise en marché. En éliminant les étapes de fabrication (masques) et leurs coûts non-récurrents, les FPGA accélèrent les multiples itérations inhérentes à tout design, pour un moindre prix. Les cycles de design sont beaucoup plus simples car c'est au niveau logiciel que s'opèrent le placement et routage. En combinant les meilleures propositions d'un système basé sur processeur ou d'un ASIC, les FPGA ont offert une grande flexibilité et la possibilité d'un prototypage rapide pour les ingénieurs et scientifiques de toutes les expertises. Des semaines de travail se transforment littéralement en heures, une caractéristique très intéressante pour des objectifs industriels.

Néanmoins, malgré plusieurs progrès dans la description de systèmes logiques à un degré supérieur d'abstraction, le design matériel de systèmes complexes reste un travail de spécialistes. En effet, les ressources disponibles augmentent à chaque nouvelle mouture, menant à des architectures de plus en plus élaborées et à une gestion de plus en plus difficile. Les concepteurs ont donc maintenant accès à des ressources qui semblent gratuites tellement il y en a, mais la complexité de les utiliser efficacement demande énormément de travail et d'expertise, et il est possible qu'on n'arrive plus à satisfaire aux demandes du marché. Le FPGA proposerait alors des performances et un pouvoir de calcul inégalés, mais serait difficilement utilisable car on n'arriverait pas à extraire de façon simple et efficace toute cette puissance. Prenons comme exemple le cas où on désire faire la racine carrée sur une grande étendue de valeurs. Initialement, on peut utiliser une ALU et faire un algorithme qui va réaliser la racine, et faire passer chacune des valeurs à travers ce flot de données. Cependant, avec les ressources nécessaires disponibles, il serait possible d'avoir plusieurs ALU en parallèle qui

font les calculs sur plusieurs valeurs à la fois. Qui plus est, un système pourrait s'assurer de ne pas recalculer une valeur déjà connue. On pourrait même développer des algorithmes beaucoup plus complexes pour avoir de meilleures précisions. Si les ressources sont disponibles, les possibilités sont nombreuses mais encore faut-il savoir s'en servir. En regardant les nouvelles tendances qui consistent à former des réseaux de plusieurs FPGA, il y a un besoin urgent de non seulement pouvoir exploiter de façon efficace les ressources mais également de gérer le niveau de communication que cela demande.

Le but ultime serait d'avoir les performances offertes par du matériel dédié avec la simplicité et la flexibilité du développement logiciel. Plusieurs méthodes tentent de combler ce besoin, dont les logiciels de synthèse à haut-niveau qui prennent comme point de départ un langage tel que le C, C++ ou *SystemC*. La description algorithmique est alors analysée afin de créer une architecture qui se transpose au niveau d'un langage de description matérielle tel que le VHDL, qui est à son tour synthétisé. Hélas, ces outils offrent très peu de flexibilité et l'éventail des applications supportées est faible. En effet, on réalise rapidement que pour que ces outils donnent de bons résultats, il faut manier le code en connaissance de cause, c'est-à-dire en sachant comment la transposition se fera. Cette notion va à l'encontre de ce qu'on essaie d'accomplir et il serait alors plus profitable de décrire directement le circuit dans un langage matériel. Une autre approche est de réutiliser des modules fonctionnels d'une bibliothèque de modules IP (*Intellectual Properties*). Ces blocs peuvent représenter des entités complexes telles que des processeurs et des routeurs ou des concepts plus simples comme une ALU ou une FIFO. Puisqu'ils ont déjà été utilisés, vérifiés et documentés, il est tout à fait sensé de les incorporer dans un design. Bien que cette solution semble intéressante, elle comporte son lot d'embûches. Premièrement, les prix pour accéder à ce genre de bibliothèques sont élevés, ce qui peut causer problèmes pour de petites entreprises. De plus, certains fournisseurs ne voudront pas fournir le code source par soucis de protection de leurs propriétés intellectuelles. Finalement, l'intégration d'un système complexe comportant plusieurs modules IP de sources différentes est un énorme défi qui nécessitera un expert en design matériel.

Bien que plusieurs applications pourraient bénéficier d'une implémentation matérielle, les concepteurs qui ont l'expertise et la connaissance pour le faire sont peu nombreux. Il y a un vrai besoin pour des méthodes de transposition d'applications et d'algorithmes sur FPGA, mais les solutions restent rares. La taille des FPGA augmente de manière exponentielle. Il est intéressant de remarquer que plusieurs sociétés qui font de la conception électronique nécessitent des puces avec des I/O (*Input/Output*) de haute performance. Les tableaux 1.1 et 1.2 montrent les vitesses d'entrées/sorties et les ressources en fonction du modèle de FPGA.

Tableau 1.1 Ressources pour la famille Stratix d'Altera

| Nom | Vitesse des I/O | ALM |
|---|---|---|
| Stratix 5 5SGXBB | 28,05 Gb/s | 359200 |
| Stratix 4 EP4SGX530 | 11,3 Gb/s | 212480 |
| Stratix 3 EP3SE260 | 1,6 Gb/s | 102000 |

Tableau 1.2 Ressources pour la famille Virtex de Xilinx

| Nom | Vitesse des I/O | Slices |
|---|---|---|
| Virtex 7 XC7VH870T | 28,05 Gb/s | 136900 |
| Virtex 6 XC6VHX565T | 11 Gb/s | 88560 |
| Virtex 5 XC5VFX200T | 6,5 Gb/s | 30720 |

On y voit que plus les I/O sont performants, plus le nombre de ALM (Altera) ou de *Slices* (Xilinx) augmente. Ces ressources sont souvent laissées libres car il y en a toujours de plus en plus, comme c'est le cas pour le partenaire industriel du projet. Celui-ci aimerait fournir un accès à cette puissance de calcul pour faire du pré-traitement sur les données entrant dans leur carte d'acquisition. L'objectif du projet est de fournir une architecture qui permettrait non seulement d'utiliser efficacement les ressources de plus en plus présentes mais de camoufler les détails de l'architecture matérielle afin de ne donner qu'un accès logiciel. Ainsi, un utilisateur qui ne connait pas grand chose au design à bas niveau peut venir programmer ses applications et exploiter le parallélisme convoité, sans avoir à se soucier des questions de communication, de synchronisation et d'implémentation. Le SDPFGA (*Software Defined FPGA*) développé dans le cadre de cette recherche propose la configuration du FPGA à deux niveaux d'abstraction. L'idée est la suivante : au niveau matériel (premier niveau de configuration), nous réaliserons un treillis de calcul constitué de divers blocs permettant de gérer le chemin de contrôle et le chemin de données. Cette architecture sera fixe, c'est-à-dire qu'il faudra simplement spécifier sa grandeur, extensible dans les 2 dimensions, avant d'en faire la synthèse. Par la suite, au niveau logiciel, il sera possible de venir y instancier des algorithmes (deuxième niveau de configuration). L'enjeu sera de fournir un circuit performant dont le pipeline assurera le meilleur débit possible en sortie, tout en parallélisant les calculs. Typiquement, une entreprise livrerait une carte FPGA déjà configurée à bas niveau et offrirait à ses clients des possibilités de configuration de plus haut niveau, et ce de façon dynamique. La technologie deviendrait beaucoup plus accessible pour des utilisateurs de cartes numériques. Également, des jours de développement se transformeraient en heures à l'aide du produit final. Cette structure à flot de données s'inscrit dans le domaine du traitement numérique du signal.

Les contributions de ce travail pour la science sont les suivantes :

- Conception d'une nouvelle architecture exploitant deux niveaux d'abstraction différents dans le but de faire du développement matériel avec la flexibilité d'une implémentation logicielle ;
- Développement d'une nouvelle méthodologie ou protocole pour faciliter l'implémentation d'algorithmes sur la dite architecture ;
- Élaboration d'une méthode, à travers ce treillis, qui permettra aux implémentations futures sur FPGA de suivre la courbe exponentielle de complexité et de rendre la technologie toujours plus accessible ;
- Utilisation efficace des ressources de plus en plus abondantes afin de fournir des performances toujours croissantes.

Ce présent document étalera les différentes étapes du projet. En premier lieu, une revue de littérature s'impose afin de bien cerner le sujet et de voir ce qui existe déjà en terme d'architectures reconfigurables. Les notions de largeur de grains sur une topologie matérielle ainsi que de flots de données serviront à introduire les NoC (*Network-On-Chip*). Ce type de circuit formera une importante base pour l'élaboration de notre solution. On analysera par la suite les treillis les plus populaires dans la communauté scientifique. Cette étape d'étude et d'analyse de l'état de l'art est indispensable afin de dégager les enjeux architecturaux dont il faudra tenir compte, dans le but de fournir une solution viable. Par la suite, cette solution sera décrite et expliquée en détail. Le treillis proposé est constitué de 3 modules distincts : des *SALU* (*Shared ALU*), des *Banques de FIFO* ainsi que des *Machines à états*. Cette portion du document permettra de bien comprendre toutes les subtilités du circuit. En dernier lieu, nous analyserons les différents résultats expérimentaux. On regardera l'implémentation d'un filtre FIR pour montrer la simplicité d'utilisation du produit final. Il sera également intéressant de tester la reconfiguration dynamique de ce SDFPGA et d'en préciser les bienfaits. En terminant, notons que cette recherche fut le sujet d'un article publié dans la conférence ISCAS 2012 (voir Allard *et al.*, 2012). Cette publication peut servir de complément à ce mémoire.

# CHAPITRE 2

# REVUE DE LITTÉRATURE

## 2.1 Introduction

Le traitement numérique du signal est un vaste domaine duquel découle plusieurs avancements technologiques. En effet, les poussées techniques en ingénierie des ordinateurs proviennent largement du besoin continu d'augmenter la puissance de calcul. Initialement, les algorithmes étaient pris en charge par des ordinateurs standards. Les ASIC et les processeurs *DSP* construits sur du matériel dédié prirent rapidement leur place sur le marché. De nos jours, la demande accrue d'applications traitant de grands flots de données en temps réel pousse la recherche vers de nouvelles avenues telles que des microprocesseurs à usage général plus puissants et les FPGA (voir Stranneby et Walker, 2004, p.241). Ces derniers représentent la plateforme matérielle étudiée dans le présent document, et l'état de l'art qui est présenté en dérive.

En premier lieu, nous étudierons les *Coarse-Grained Reconfigurable Architecture* (CGRA). La notion de pouvoir contrôler directement des composants de plus haut-niveau permettra de mieux gérer le tissu de calcul. Par la suite, nous analyserons les principales notions et l'historique des architectures à flot de données. Ceci nous mènera au *Network-on-Chip* (NoC) et à l'oscultation des différentes topologies de tissu de calcul. Finalement, les treillis popularisés, plus particulièrement le RaPiD, le PipeRench, le RAW et le MorphoSys, en plus d'architectures plus actuelles, serviront de modèles pour l'élaboration de notre propre design. Il sera alors possible de prévoir les limitations de la solution proposée afin de les contrer lors de l'élaboration du design.

Il va sans dire que l'architecture voulue provient d'une série de fonctionnalités pré-établies selon les besoins du projet. Le défi sera de prendre en compte les différents résultats de ce chapitre et de les appliquer sur la solution.

## 2.2 CGRA : *Coarse-Grained Reconfigurable Architecture*

Le terme "granularité" est largement utilisé dans le domaine de la science. Il représente la mesure par laquelle un système est décomposable en plus petites parties. Par exemple, un kilomètre divisé en mètre a une granularité plus fine qu'un mètre divisé en pieds. Ainsi

donc, il est possible de classifier toutes les architectures informatiques selon leur granularité. Les deux grandes familles sont les suivantes : *fine-grained* (à grains fins) et *coarse-grained* (à grains grossiers). La granularité d'un tissu reconfigurable est définie par la complexité de sa plus petite unité fonctionnelle, basée sur la largeur des données et des capacités de calcul (voir Svensson, 2009). Ainsi, la technologie FPGA est considérée à grains fins, comme le montre la figure 2.1. En effet, les FPGA sont faits d'un tissu de blocs logiques (contenant des tables de vérité de petite taille en comparaison à un processeur), de ressources de routage programmables et d'éléments d'entrées/sorties.

Assez rapidement, il fut découvert que les FPGA introduisaient plusieurs désavantages pour effectuer des tâches de calculs complexes. Premièrement, les opérations sur de grands chemins de données devaient être composées de plusieurs unités de calcul opérant sur des symboles binaires. À cela s'ajoutait le coût du routage et des interconnexions, créant ainsi une solution relativement peu efficace sur la densité de silicium utilisée. Par la suite, il faut remarquer que le développement d'applications sur FPGA ressemble beaucoup à celui d'un design VLSI de par leur programmation au niveau logique. Donc, l'implémentation d'algorithmes provenant d'un langage haut-niveau est difficile en comparaison avec une compilation sur un processeur standard car la granularité du FPGA ne correspond pas aux opérations du code source (voir Todman *et al.*, 2005). La méthode standard consiste encore à utiliser un langage de description matérielle (VHDL, Verilog) et requiert ainsi une expertise qui est moins répandue que la programmation en logiciel. L'objectif principal du projet est de réduire l'incidence de

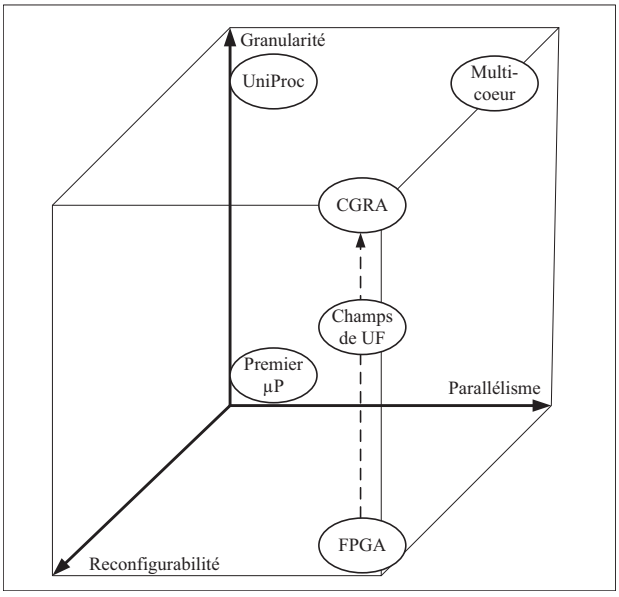


Figure 2.1 Positionnement des technologies en fonction de 3 critères

ces problèmes par l'entremise d'une architecture à plus gros grains, contrôlée de manière logicielle.

Les architectures à gros grains tentent d'éliminer ces inconvénients par l'utilisation de chemins de données à plusieurs bits et d'opérateurs plus complexes. Puisque leurs blocs logiques sont optimisés pour de grands calculs, les CGRA vont opérer plus rapidement et consommer moins d'espace que la même structure créée à partir de cellules plus petites. Malgré tout, le goulot d'étranglement se situe dans le chemin de contrôle. En effet, bien que la particularité du chemin de données soit attrayante, il s'agit aussi de la cause du surcoût important au chemin de contrôle. Le routage d'un grand nombre de bits de configuration à travers les architectures est un désavantage majeur (voir Park *et al.*, 2009; Compton et Hauck, 2002). Ainsi, les CGRA sont normalement constitués d'unités fonctionnelles (UF) qui permettent de faire directement des opérations telles que des additions, soustractions, etc. On peut les voir comme des ALU interconnectées entre elles à travers un réseau.

Historiquement, la technologie FPGA a également tenté de suivre la tendance des CGRA. À titre d'exemple, la cellule logique du *FLEX10K* (Altera) est une architecture à grain fin, mais plus gros que celui du *6200*. Elle consiste en une table de vérité à 4 entrées, avec une bascule et une chaîne de propagation de la retenue. Sur le Stratix III, qui est plus récent, chacune des ALM (*Adaptive Logic Module*) possède une table de vérité à 8 entrées, en plus de deux additionneurs, deux registres et un routage pour la chaîne de retenue (voir ALTERA, 2007). Puisque nous travaillons dans un domaine reconfigurable, il est possible d'avoir des largeurs de données différentes(une multiplication sur 5 bits, une addition sur 18 bits, etc.), ce qui n'est généralement pas directement supporté sur un processeur. Pour une largeur excédentaire, celui-ci devrait rajouter des étapes pour quérir, décoder et exécuter des instructions additionnelles, alors qu'un FPGA peut tout réaliser en une seule étape, tant que les ressources sont disponibles.

De nos jours, les avancées technologiques permettent d'instancier des éléments à gros grains à travers la logique et le routage à grains fins des FPGA, tels que des multiplieurs, des mémoires, des processeurs, etc. Ce progrès offre un énorme intérêt en rajoutant l'aspect "reconfigurabilité" que l'on retrouve en figure 2.1. Une considération importante lors de l'ajout de ces éléments au FPGA est l'interface entre les ressources à grains grossiers et à grains fins. Si elle n'est pas assez flexible, l'utilité de l'élément ajouté en sera réduite puisque les connexions seront coûteuses. Il deviendrait presque impossible d'en faire le routage. Cependant, une trop grande flexibilité entraînerait des coûts en surface qui pourraient nuire à la

performance lorsqu'on implémentera des applications ne nécessitant pas le module en question. Des chercheurs (voir Chi *et al.*, 2008) se sont penchés sur ce phénomène par l'entremise d'unités fonctionnelles réalisant des calculs arithmétiques en virgule flottante. Leurs conclusions sont que les UF devraient être positionnées au centre de la puce, avec les entrées et les sorties distribuées uniformément sur le pourtour. De plus, fait intéressant dans le cadre de notre projet, ils affirment que leur implémentation la plus efficace était celle pour une répartition en carré des unités fonctionnelles. Il sera effectivement discuté plus loin de la disposition de nos propres modules formant le tissu. Bien que leur travail ait été effectué sur du calcul en virgule flottante, les auteurs prétendent que leur analyse vaut également pour d'autres types de blocs embarqués.

Avec cette hausse de complexité vient la nécessité d'outils pour faire le placement des différents modules ou UF sur les cellules du FPGA. Plusieurs techniques existent en littérature, et ressemblent toutes à celle qui est décrite par Parvez (voir Parvez *et al.*, 2008). Les auteurs utilisent un flux logiciel pour venir instancier une *netlist* sur le FPGA et affiner la position des différents modules. Pour ce faire, ils utilisent une grille avec des cellules de tailles diffé-

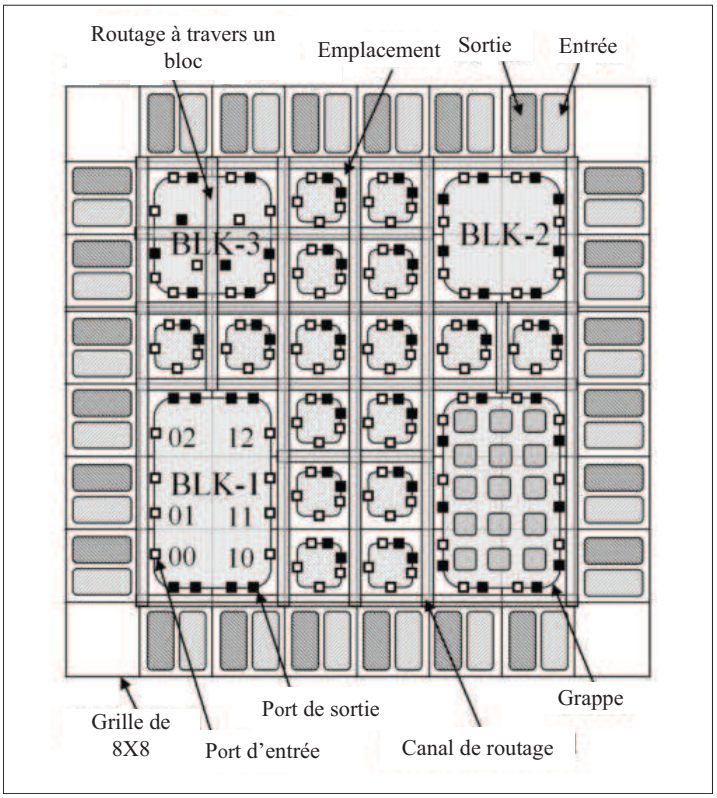


Figure 2.2 FPGA à grains grossiers

rentes (voir figure 2.2). Un algorithme procure le placement ayant la somme minimale des demi-périmètres des *bounding box* (aire rectangulaire minimale qui contient toutes les entrées et les sorties du *net*) en déplaçant les instances d'un site à un autre. L'article conclut en soulignant que le temps nécessaire pour optimiser le placement est 8 fois plus rapide avec leur technique.

Parmi les nombreuses configurations de type CGRA existantes en littérature, MORA (*Multimedia Oriented Reconfigurable Array*) démontre bien les avantages d'aller vers un design à grains grossiers (voir Lanuzza *et al.*, 2007; Chalamalasetti *et al.*, 2009). Contrairement aux autres architectures (dont certaines seront étudiées dans les prochaines sections), celle-ci est une plateforme générale et n'est pas optimisée pour donner le meilleur débit pour un certain type d'applications. Un des objectifs lors du design était de pouvoir instancier le plus facilement possible les algorithmes désirés. Cet aspect est en fait essentiel à la survie du projet car s'il devient trop complexe d'utilisation, il sera rapidement dépassé par d'autres architectures. La figure 2.3 montre le design en deux dimensions (à gauche) ainsi que le circuit représentant l'unité de calcul (à droite). Chacune des cellules reconfigurables possède une unité de calcul de 8 bits, une mémoire 256x8 à double ports et un contrôleur central qui contient les instructions. À titre d'exemple, l'unité de calcul est constituée de sous-modules très simples qui permettent de faire des opérations arithmétiques et logiques sur deux opérandes. Un aspect

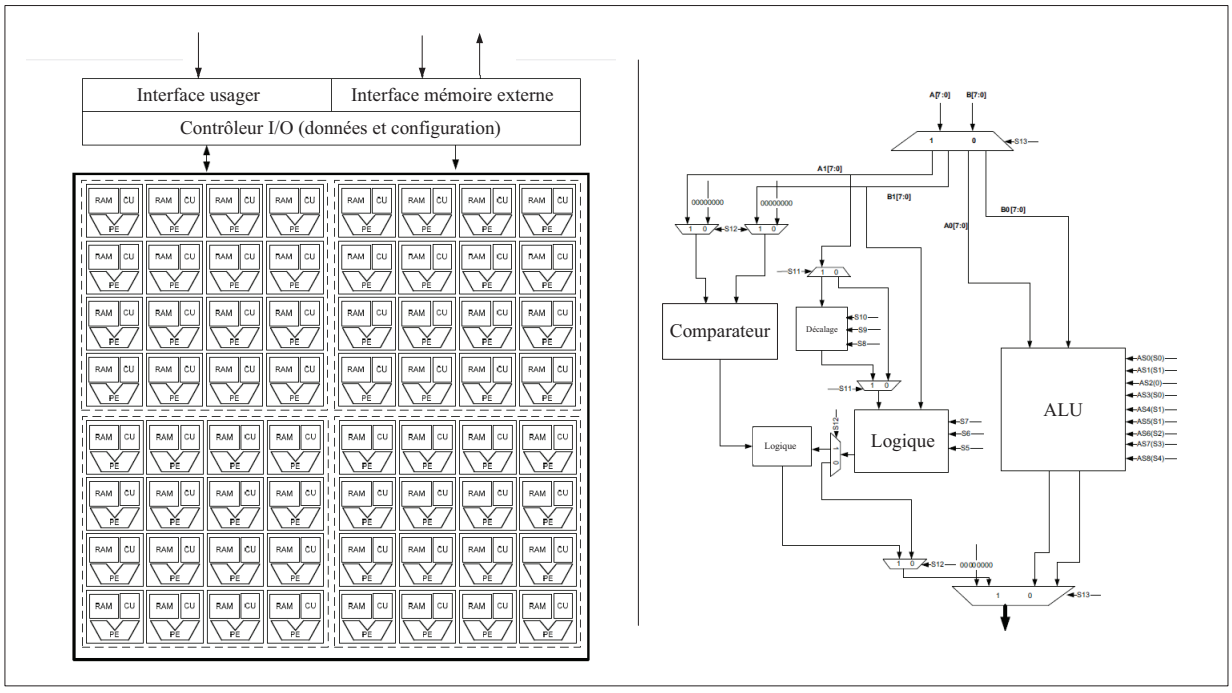


Figure 2.3 Architecture MORA et son unité de calcul

important de leur design exige que les calculs soient réalisés près des blocs de mémoire. On préconise ici l'utilisation de petites mémoires décentralisées pour que chacune des cellules puisse travailler de façon indépendante. De plus, leur mot d'instruction est plus long pour permettre de faire des sauts dans le programme, ce qui est un grand avantage sur le contrôle des instructions. Cependant, comme nous l'avons cité précédemment en terme de désavantage des CGRA, les différentes cellules doivent propager le contrôle entre elles. Pour une multiplication matricielle, des résultats à 166MHz démontrent la grande efficacité de l'architecture à grains grossiers.

En conclusion, plus la granularité est fine, plus le potentiel de parallélisme (et de vitesse) est grand, mais plus le sont également les coûts en terme de synchronisation, de routage et d'interconnexions. D'un autre côté, pour une architecture à grains trop gros, les performances pourraient souffrir d'un déséquilibre des opérations. Il est donc primordial de trouver le juste milieu.

## 2.3 Architecture à flot de données

La motivation originale pour les architectures à flot de données fut l'exploitation massive du parallélisme. C'est dans les années 1970 que Dennis et Misunas (voir Dennis et Misunas, 1975) développèrent le modèle de calcul à flot de données. À l'époque, les processeurs étaient développés sous les notions de Von Neumann qui propose une architecture à flot de contrôle. Il y a donc un minimum de ressources au niveau calcul et mémoire pour permettre à une entité tierce (un contrôleur) de gérer le tout selon une série d'instructions (programme). L'instruction qui sera traitée dépend uniquement du compteur de programme, ce qui est un grand inconvénient. On peut imaginer la situation où deux opérandes-soeurs sont disponibles mais ne sont pas utilisées. Les architectures à flot de données tentent de remédier à la situation. En effet, il n'y a pas de compteur de programme ni de contrôleur central car l'exécution des instructions est entièrement synchronisée sur la disponibilité des opérandes. On utilise plutôt des graphes permettant de montrer la relation entre les différentes opérations. L'exemple de la figure 2.4 montre l'ordonnancement pour le calcul de la moyenne :

$$\frac{x+y+z}{3}$$

Et de l'écart-type :

$$\sqrt{\frac{x^2+y^2+z^2}{3} - (\frac{x+y+z}{3})^2}.$$

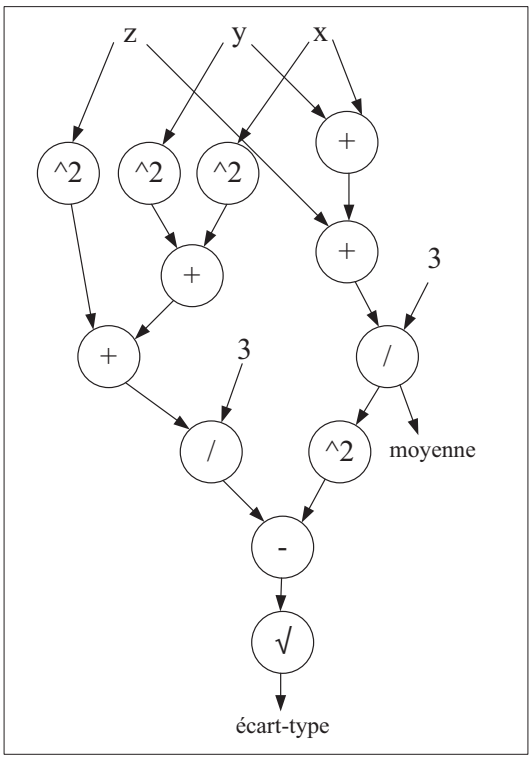


Figure 2.4 Graphe d'ordonnancement

À priori, cette technique est celle qui permet le meilleur parallélisme possible car pour le meilleur agencement, il est impossible d'aller plus vite vu la dépendance des données. Les données sont considérées comme étant des jetons qui sont consommés à travers les opérations pour en créer de nouveaux. Deux techniques majeures (voir Johnston *et al.*, 2004) ont été utilisées à travers les différentes recherches dans le domaine :*token model* et *structure model*. La première propose que les jetons s'accumulent aux sorties des opérations pour ensuite être repris dans cet ordre, alors que la deuxième définit les jetons résultats dans une mémoire avec un accès aléatoire de sorte à pairer les jetons de même "couleur". Bien que cette dernière technique soit attrayante, il devient très difficile de sauvegarder ces jetons. Il faut en effet trouver une façon de pouvoir retrouver ceux qui sont associés. Cette technique ne fut pas reconduite dans les projets qui découlèrent des architectures à flot de données. Parmi les architectures les plus reconnues, celle du MIT (voir Dennis, 1980) utilise la technique de *token model* en ayant qu'un seul jeton par arc (sortie d'une opération). La figure 2.5 montre le schéma pour le *structure model* proposé par l'Université de Manchester (voir Watson, 1979). À la sortie des opérations réalisées par les unités fonctionnelles, les jetons sont mis dans une file pour qu'un module d'agencement s'occupe de les renvoyer pour de nouvelles instructions.

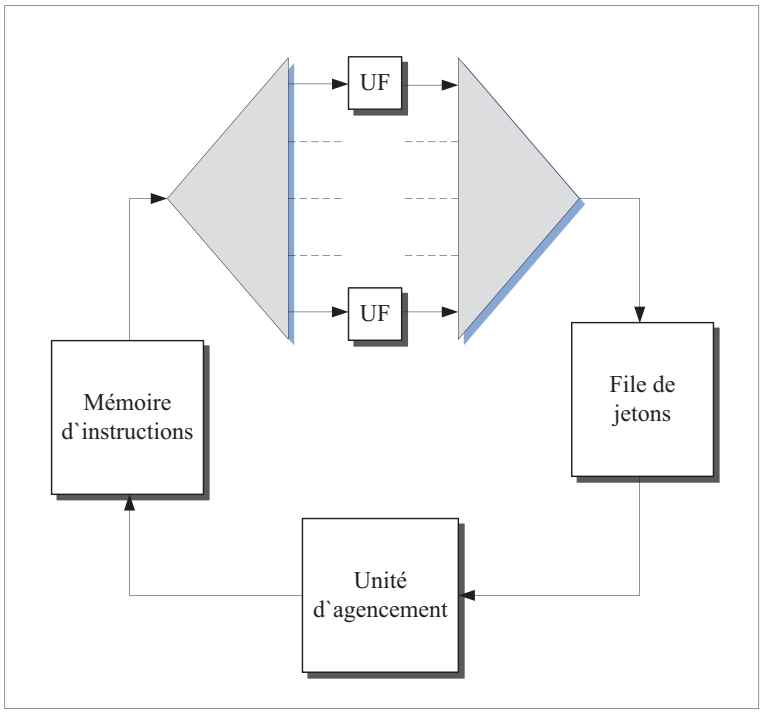


Figure 2.5 Architecture de l'Université de Manchester

Il y a eu un regain pour les architectures à synchronisation par les données dans les années 1990 lorsqu'on réalisa qu'un hybride entre une architecture Von Neumann et à flot de données procurait de nouveaux niveaux de performance. En effet, le défi résidait au niveau de la granularité. En 1995, dans *Studies on Optimal Task Granularity and Random Mapping* (voir Sterling *et al.*, 1995, p.349-365), Sterling explore les performances de différentes architectures à flot de données pour fournir le graphe de la figure 2.6. On y voit que ce n'est pas une granularité fine ou grossière qui offre les meilleures performances parallèles mais plutôt un juste milieu.

Malgré tout, plusieurs problèmes ont persisté causant le maintien de ce concept au niveau théorique. Pour un système parallèle de taille considérable, faire le routage efficace des jetons de données et d'instructions est une tâche extrêmement ardue. De plus, les itérations que l'on retrouve fréquemment dans les algorithmes ne se transposent pas très bien dans la plupart des architectures décrites en littérature. Avec l'arrivée de la programmation structurée et de la programmation orientée-objet, l'idée d'un développement logiciel sans l'utilisation de structures de données est impensable. Toutefois, certaines recherches tiennent en compte ces structures, mais ce n'est pas largement répandu.

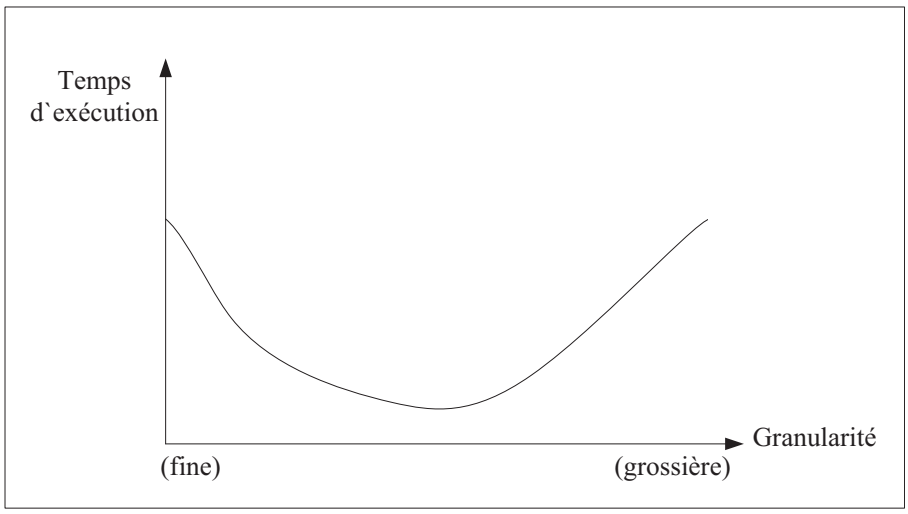


Figure 2.6 Graphe du temps d'exécution en fonction de la granularité

Bien que le processeur superscalaire éclipsa l'architecture à jetons dans les années 1990, certaines études persistent encore aujourd'hui dans le domaine. Il est maintenant possible d'avoir des milliards de transistors par puce ; il faut donc trouver une façon d'utiliser ces ressources de manière efficace. Bergeron et David (voir Bergeron *et al.*, 2005) proposèrent un langage haut-niveau, le CASM (*Channel Based Algorithmic State Machine*), permettant à des utilisateurs peu expérimentés dans le design de circuits numériques d'explorer la technologie reprogrammable qu'est le FPGA. Le langage est inspiré du procédé de machine à jetons décrit dans la présente section et supporte la récursion, procédé absent des autres langages du même genre. L'article de Thielmann (voir Thielmann *et al.*, 2011) amène une solution (*PreCoRe*) pour générer automatiquement des chemins de données. Il utilise la technique de transmission de jetons pour faire des chemins spéculatifs et gérer les chemins acceptés et refusés.

D'autres recherches tentent encore aujourd'hui d'appliquer les techniques développées dans le domaine des architectures à flot de données sur des architectures sommes toutes récentes. Comme nous avons vu précédemment, un problème majeur des CGRA réside dans les chemins de contrôle. Dans le projet actuel, nous tentons d'agencer ces deux domaines pour créer une solution plus intéressante sans toutefois s'empêtrer dans les désavantages d'une synchronisation par les données. En fait, l'étude de l'état de l'art fait ressortir ce point. Afin de toujours produire de nouvelles architectures plus puissantes sans réinventer la roue, il est primordial d'analyser les pour et contre des architectures les plus intéressantes. Park fait remarquer dans ses études des CGRA que l'ajout d'une synchronisation par les données entraine une baisse de consommation de puissance de 74% sur les mémoires d'instruction et 56% sur le chemin de contrôle (voir Park *et al.*, 2009). On peut voir un réel impact sur le domaine des architectures

à grains grossiers, et nous tenterons dans cet ouvrage d'en retirer le plus possible. Toujours dans le même ordre d'idée, l'élasticité d'un circuit représente la tolérance aux variations dans les délais de calcul et de communication (voir Carmona *et al.*, 2009). Bien que cette notion s'applique plus aux circuits asynchrones, on peut extrapoler sur les architectures à jetons et diastoliques. Ces dernières proposent des unités fonctionnelles qui communiquent uniquement par des FIFO (voir Kinsy *et al.*, 2008). Peu importe le temps que prennent les opérations, on s'assure ainsi d'avoir une très bonne élasticité. Kinsy ajoute cependant que le coût en ressource est assez élevé. L'objectif principal sera donc de permettre une communication de données par jetons tout en conservant un coût peu élevé.

## 2.4 Circuit *Network-on-Chip*

Les *System-on-Chip* (SoC) sont des circuits intégrés comprenant tous les composants d'un système électronique sur une seule puce. À titre d'exemple, un SoC peut contenir un récepteur audio, des convertisseurs numériques/analogiques, un microprocesseur, de la mémoire, de la logique arithmétique, etc. L'augmentation de l'intégration sur puce permet de réduire les coûts, la consommation de puissance et la superficie du circuit complet, tout en offrant une fiabilité supérieure. Bien entendu, le point crucial dans un SoC est la communication interne. Historiquement, les ressources de calcul ont toujours été coûteuses, alors que celles utilisées à fin de communication furent bon marché. Cependant, la réduction des circuits électroniques vint inverser les rôles. La puissance de calcul devient de plus en plus abordable alors que la communication rencontre des limites physiques fondamentales. Puisque les "fils" de communication sur puce ne se réduisent pas de la même façon que les transistors, il y a un besoin d'équilibre entre le chemin de données et de contrôle. Bien que le temps nécessaire aux communications globales ne change pas, celui utilisé sur un calcul local ne cesse de diminuer. Cette différence drastique va faire que le réseau d'interconnexions d'un SoC va largement dominer la performance de celui-ci (voir Sylvester et Keutzer, 2000; Giefers et Platzner, 2007).

La solution initialement adoptée au problème de communication (dans les années 1990) fut de créer un mélange entre des communications point-à-point et par bus propre à chaque architecture. Malheureusement, ces techniques entrainent des difficultés pour un système de taille moyenne et plus. C'est ici qu'entrent en jeu les *Network-on-Chip* (NoC), qui constituent la tentative de créer un système de communication intra-SoC. En effet, les notions de réseautage sont utilisées pour créer une méthode plus générale comprenant des noeuds de routeurs à travers l'architecture, interconnectés par un réseau de communication. La représentation la plus simpliste d'un NoC se retrouve en figure 2.7. Les adaptateurs réseau représentent

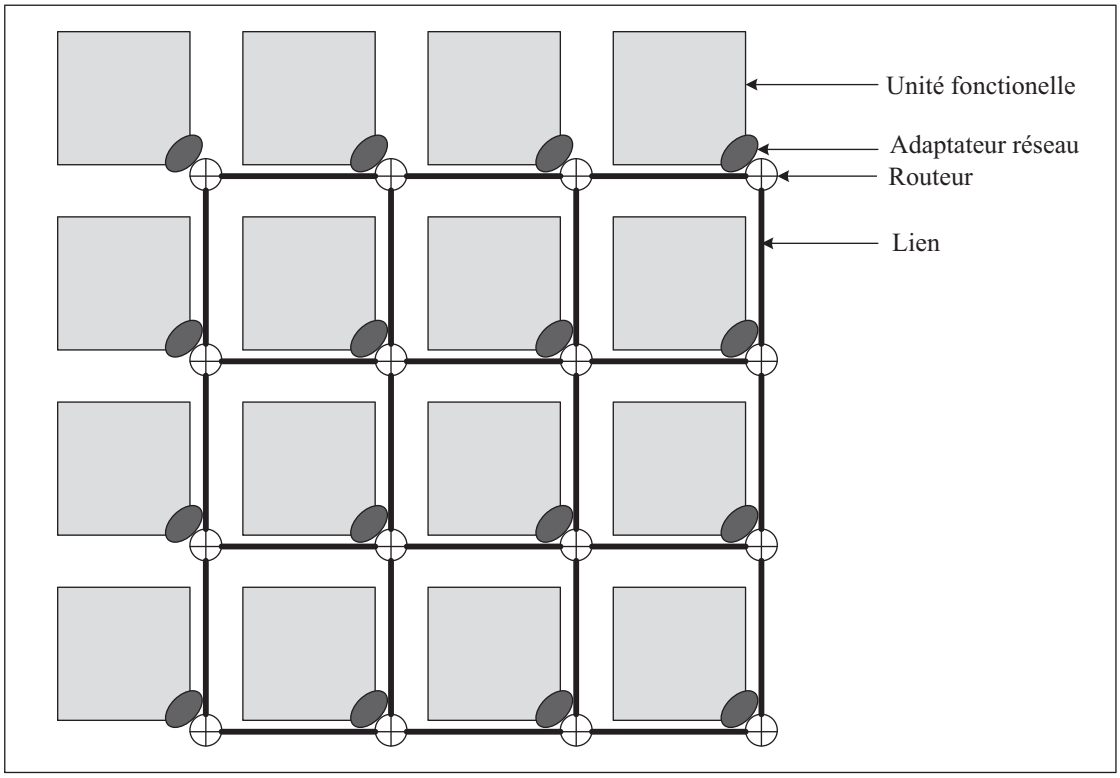


Figure 2.7 Modèle de base d'un NoC

l'interface par laquelle les unités fonctionnelles sont reliées au réseau. Ces blocs font le découplage entre le niveau dit de "calcul" et celui de "communication". Les routeurs implémentent la stratégie de communication des données. Le protocole en question varie selon le design. Par exemple, on pourrait vouloir avoir des communications de la gauche vers la droite et du haut vers le bas uniquement. Finalement, les liens représentent la bande passante entre les routeurs. Ils peuvent être constitués d'un ou de plusieurs canaux (voir Bjerregaard et Mahadevan, 2006). Au-delà de ce modèle, le terme *Network-on-Chip* est utilisé à plusieurs saveurs dans la littérature. Il englobe maintenant toutes les techniques de propagation de mots de données et de contrôle à travers un SoC. Pour le reste du présent ouvrage, cette définition de NoC est celle utilisée.

La caractéristique la plus importante d'un *Network-on-Chip*, outre le protocole de communication dont nous discuterons à travers les différentes architectures connues, est sa topologie. En effet, il existe une multitude de façons de brancher les différents modules d'un NoC, chacune comportant son lot d'avantages et d'inconvénients. Nous discuterons ici de certaines d'entre elles afin de voir les plus utiles. Différentes topologies se retrouvent sur la figure 2.8.

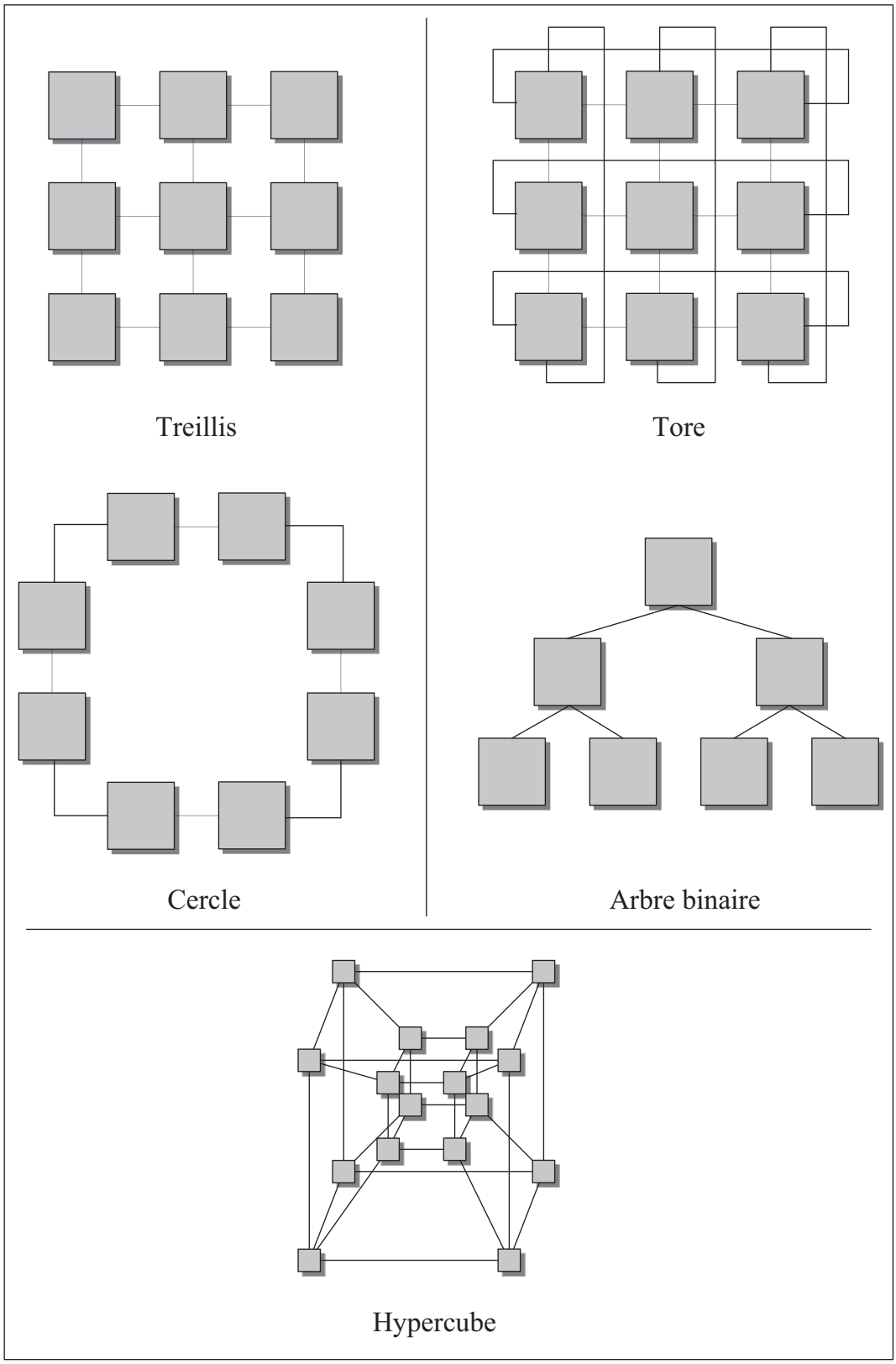


Figure 2.8 Différentes topologies de *Network-on-Chip*

Le treillis (*mesh*) est l'architecture la plus répandue à travers les NoC. Dans celle-ci, chacune des unités fonctionnelles est reliée à ses voisins immédiats, sans plus (l'image simplifiée ne décrit pas la technique de communication, uniquement les liens possibles entre UF). Puisque les connexions locales entre les UF sont petites, la surface consommée par les fils est négligeable et il est donc réaliste de dire que la complexité d'une implémentation en treillis dépend grandement du nombre d'unités fonctionnelles. De plus, il est possible d'opérer à de très grandes vitesses en conséquence du court délai de propagation entre les UF (voir Parhami, 1999). Si l'un des noeuds du treillis fait défaut, il est toujours possible de le contourner ; le transfert de données n'est pas interrompu. L'ajout d'unités fonctionnelles est également très simple sans avoir à modifier l'agencement (facilement extensible dans les 2 ou même 3 dimensions si nécessaire). Cependant, le treillis suppose que la matrice FPGA est intrinsèquement homogène, ce qui n'est pas le cas (emplacement des ressources). Il devient difficile d'implémenter un vrai treillis avec des UF identiques. L'emplacement des I/O dans certaines régions du FPGA peut également créer un goulot d'étranglement. Finalement, on pourrait voir une perte de performance car plusieurs UF deviendront redondantes pour une bande passante moyenne (voir Devaux *et al.*, 2009). Il est important de signaler que les "tuiles" du treillis ne sont pas nécessairement toutes les mêmes. Il va sans dire que différents agencements peuvent permettre de contrer les problématiques.

La tore possède plusieurs des caractéristiques du treillis, avec l'ajout d'un lien entre les UF situées aux extrémités. Ces liens peuvent rajouter de longs délais qui nuiront grandement aux débit possibles dans l'architecture. Il est cependant possible de plier une tore de sorte à n'avoir que de courts liens locaux. L'implémentation sur FPGA en est cependant plus difficile et une bonne stratégie de placement et routage est prescrite (voir Saneei *et al.*, 2006).

La topologie en cercle (*ring*) est intéressante car le protocole de communication est assez simple ; chaque noeud intermédiaire retransmet l'information de la source vers la destination, en modifiant ou non le contenu. On peut donc atteindre de très hauts débits. Effectivement, en prenant un nombre égal d'unités fonctionnelles, le cercle aura toujours le nombre de liens le plus petit en comparaison aux autres architectures. Cependant, le mauvais fonctionnement d'un seul noeud peut causer la perte du système complet. De plus, les algorithmes plus complexes se transposent mal sur cette topologie. Elle est efficace pour des designs à petite échelle donnant un diamètre de cercle de taille raisonnable (voir Bononi et Concer, 2006).

L'arbre binaire est aussi une topologie pour laquelle certains algorithmes se transposent bien, particulièrement lorsqu'une UF n'a pas besoin d'informations des autres UF de sa propre

étage. On peut également concevoir un arbre (*fat-tree*) où les branches supérieures sont plus grosses pour devenir de plus en plus petites pour compenser un fort débit d'entrée. L'arbre n'étant pas une structure uniforme, on pourrait tirer avantage de la répartition des ressources dans un FPGA en plaçant l'étage la plus dense sur une rangée de mémoires RAM.

Notons qu'il existe plusieurs autres topologies plus complexes, comme celle de l'hypercube présenté à la figure 2.8. Cette structure est idéale pour les algorithmes récursifs, entre autres (voir Parhami, 1999). En conclusion, la topologie de treillis est celle qui est majoritairement privilégiée en littérature, malgré quelques percées qui tentent de la discréditer. En fait, ce sont les algorithmes qui définissent l'utilité d'une architecture NoC, et le but ultime est d'en combler le plus grand nombre possible avec le même design.

## 2.5 Architectures sur *Network-on-Chip*

Au fil des années, plusieurs chercheurs élaborèrent leur propre ébauche d'architectures reconfigurables. Certaines d'entre elles eurent plus de succès que d'autres et furent reprises par de nombreux ingénieurs qui les analysèrent dans le but de les caractériser. Nous discuterons à travers cette section de ces différents designs de la grande famille des NoC et des CGRA. Ce travail ne se veut pas exhaustif (en effet, une telle tâche pourrait à elle seule remplir plusieurs livres). On tentera toutefois de faire ressortir les choix architecturaux, les forces et les faiblesses des architectures les plus populaires.

### 2.5.1 RaPiD

Le but ultime de l'architecture RaPiD (*Reconfigurable Pipelined Datapath*) est de contrer les coûts élevés des implémentations de fonctions arithmétiques sur FPGA tout en allégeant le fardeau de leur programmation. Le design permet de construire des chemins de données pipelinés de manière dynamique à partir d'un mélange d'ALU, de multiplieurs, de registres et de mémoires locales. On peut donc exécuter des opérations comme celles que l'on retrouve dans un DSP en ayant un chemin de données spécifique à une application et un contrôle statique et dynamique. La portion statique détermine la structure sous-jacente du chemin de données qui reste constante pour une application donnée. Le contrôle dynamique, quant à lui, change de cycle en cycle et vient spécifier les opérandes et les opérations à effectuer. Cette technique est bien entendue utilisée sur plusieurs architectures et sera prédominante dans la nôtre.

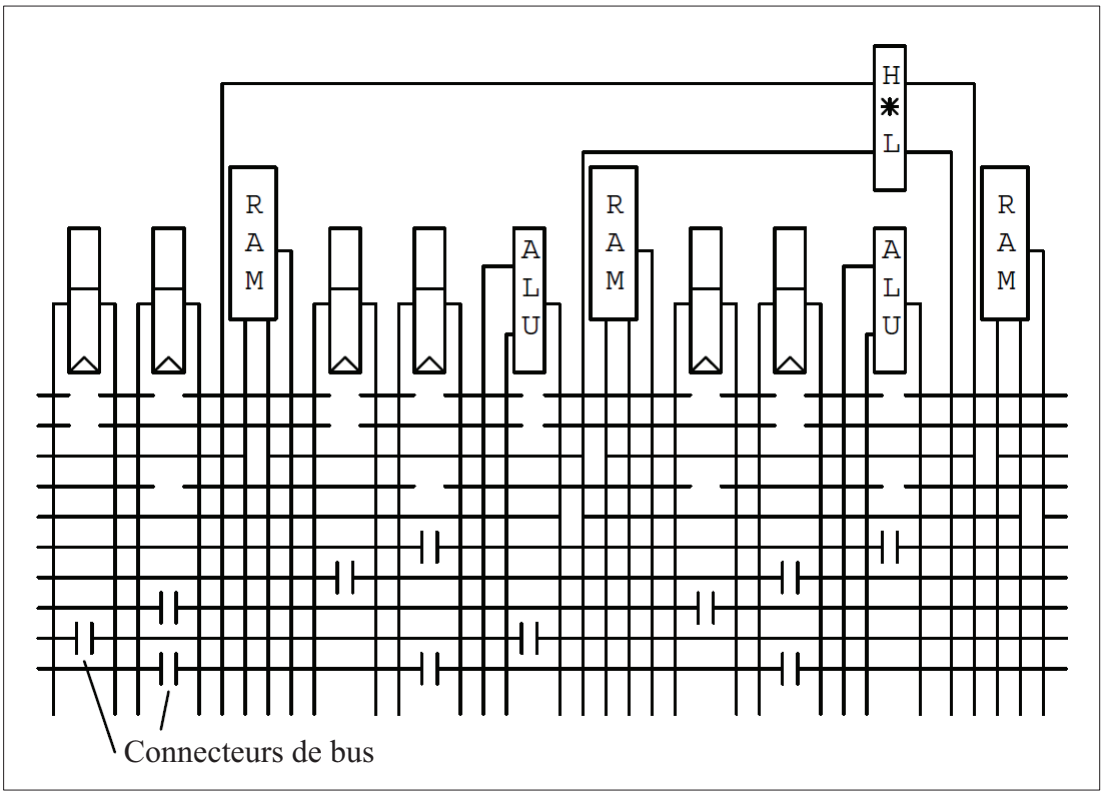


Figure 2.9 Cellule de base de l'architecture RaPiD-I

La version de l'architecture développée à l'Université de Washington est le RaPiD-I. Il existe plusieurs variantes, mais la cellule de base est fixe et se retrouve en figure 2.9. Cette cellule contient un multiplieur d'entiers (sortie de 32 bits), 2 ALU intégrées, 6 registres à usage général et 3 petites mémoires locales de 32 mots. Le tissu complet du RaPiD-I contient 16 tuiles de ce genre, bien que ces divisions soient invisibles lors de l'implémentation d'applications à travers les bus. Les différentes unités fonctionnelles sont interconnectées par 10 bus segmentés qui parcourent le chemin de données. Pour relier ces bus de manière conviviale, un connecteur de bus permet d'ajouter un étage sur le pipeline tout en dirigeant le trafic à destination. Ultimement, la partie statique est stockée dans des mémoires RAM à l'initialisation tandis que la partie dynamique est réalisée par de la logique reconfigurable.

L'architecture RaPiD a été conçu pour implémenter des applications à très haut débit. Cependant, la bande passante en entrée/sortie est bornée par les mémoires de données qui ne sont pas extensibles. Donc, le parallélisme et le gain de performance possibles en sont entachés. RaPiD n'est également pas conçue pour les tâches qui ne sont pas structurées et répétitives, et dont le flot de contrôle dépend fortement des données. Les concepteurs de l'architecture ont pris en compte que celle-ci sera associée de près à un processeur RISC

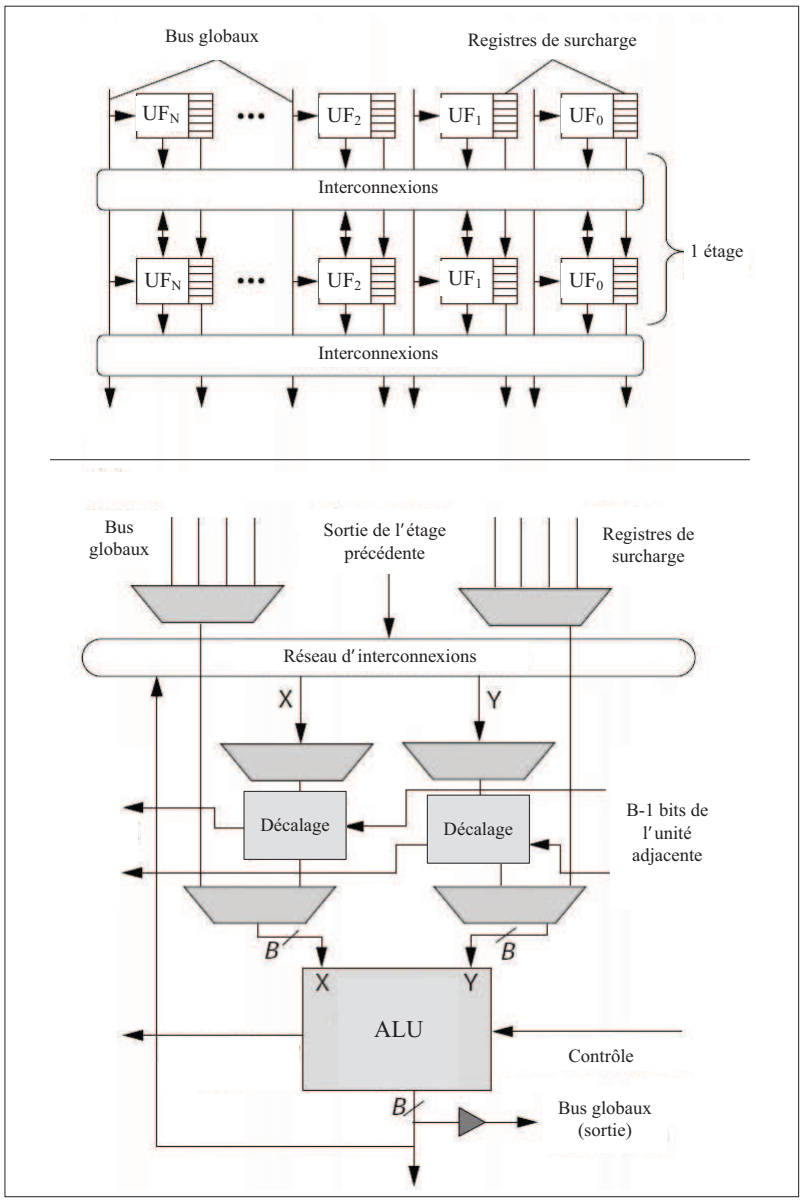


Figure 2.10 Architecture PipeRench globale (haut) et unité fonctionnelle détaillée (bas)

qui allègera le fardeau du flot. Il est également important de mentionner que c'est un NoC linéaire qui se traduit mal dans un environnement 2D. Il faut donc s'assurer que les calculs et les algorithmes puissent être exécutés sur une seule dimension (voir Ebeling *et al.*, 1996; Cronquist *et al.*, 1998; Hartenstein, 2001).

### 2.5.2 PipeRench

L'architecture PipeRench est en fait un coprocesseur pour accélérer des applications pipelinées. Son objectif est d'adapter le concept de mémoire virtuelle au matériel reconfigurable

en fournissant une architecture à plusieurs étages de pipeline reconfigurable. De ce fait, PipeRench s'appuie fortement sur la reconfiguration dynamique partielle (niveau de pipeline) et sur l'ordonnancement en temps réel du flot de contrôle et de données.

Le schéma de la figure 2.10 montre, dans un premier temps, l'architecture globale de PipeRench. Elle consiste en plusieurs étages (horizontales) composées d'un réseau d'interconnexions et d'unités fonctionnelles. Les réseaux permettent d'accéder à l'intérieur de chaque niveau ainsi que de faire des accès globaux aux autres. Au niveau global, les liens sont réalisés par 4 grands bus qui acheminent les données à travers le pipeline. Localement, c'est une *crossbar* qui permet à chacune des UF de quérir les opérandes provenant des étages précédentes ou de sa propre étage. Cependant, aucune connexion ne retourne vers un palier précédent, et chaque cycle d'une itération doit être contenu dans le même étage. Ce design ne permet donc pas de faire des boucles avec rétroaction. L'unité fonctionnelle du PipeRench est représentée au bas de la figure 2.10. Elle contient une ALU complète avec chaîne de retenue, détection du zero, etc. Comme nous pouvons le constater, les opérandes peuvent provenir d'une UF adjacente ou d'un autre étage. Le réseau complexe de bus permet également d'accéder aux entrées/sorties à partir de n'importe quel étage, mais à un fort coût au niveau du routage. Les blocs de décalage offrent la possibilité de faire de la logique ou de l'arithmétique sur de larges mots. Bien que cette approche est intéressante et qu'elle offre une fréquence de fonctionnement de 120MHz, elle souffre du même problème que la RaPiD. En effet, elle vise l'accélération des gros calculs très réguliers à travers un pipeline profond unidimensionnel. Elle n'est pas appropriée pour soutenir des applications en 2 dimensions qui sont très fréquentes dans le traitement multimédia (voir Goldstei *et al.*, 1999; Goldstein *et al.*, 2000).

### 2.5.3 RAW

RAW (*Reconfigurable Architecture Workstation*) fut développée au *Massachusetts Institute of Technology* (MIT) vers la fin des années 1990. L'idée était alors de procurer une architecture pour faire des calculs parallèles composés de plusieurs tuiles identiques connectées entres elles selon le principe du "voisin" (haut, bas, gauche, droite). Chacune de ces tuiles formant un treillis pourrait effectuer des calculs et posséderait sa mémoire (modèle de mémoire distribuée). Le design est constitué de façon à ce que le compilateur (niveau logiciel) ait toute la responsabilité de faire l'ordonnancement dans le treillis. Ceci entraîne cependant un problème lorsqu'il y a beaucoup de synchronisations nécessaires entre les différentes unités de traitement.

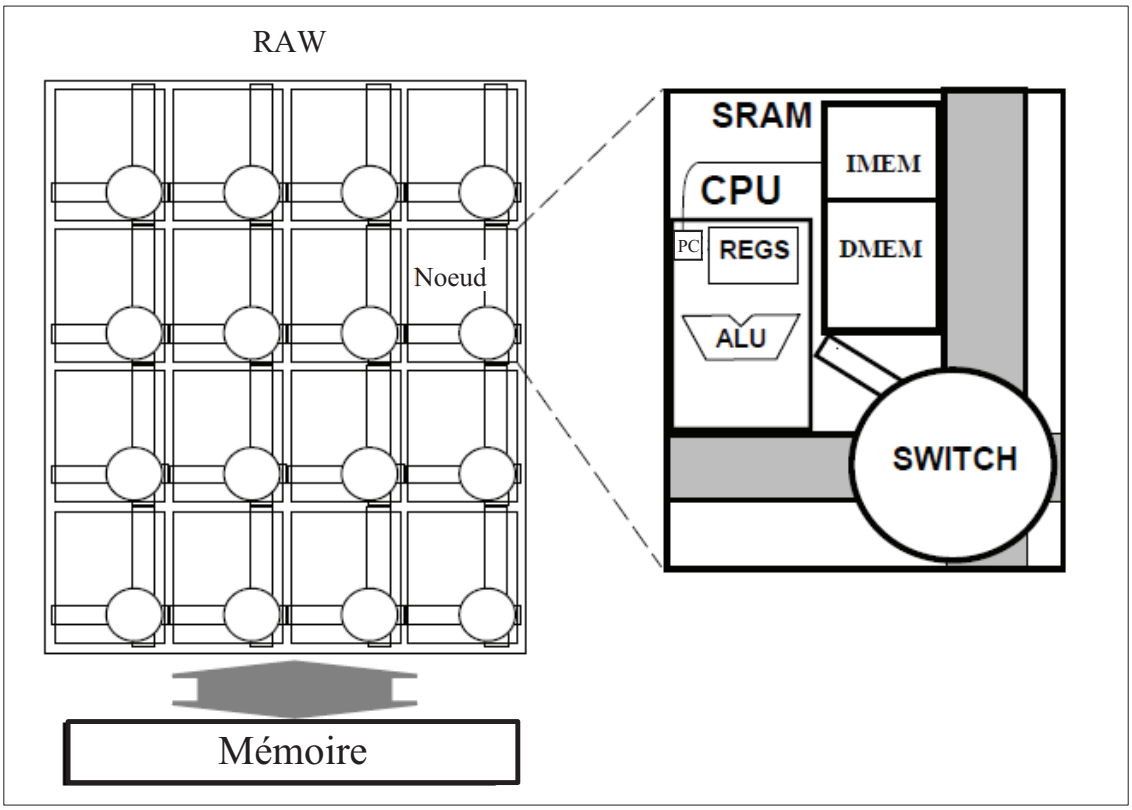


Figure 2.11 Architecture RAW

Nous pouvons voir, sur la figure 2.11, la vue d'un système typique RAW connecté à une mémoire externe. Ce prototype contient 16 tuiles, soit un treillis 4X4. Chacune des tuiles contient un semblant de processeur RISC (une ALU, des registres ainsi qu'un compteur de programme), deux mémoires SRAM de 32 Kb pour les instructions et pour les données, et une *switch* programmable permettant de faire des connexions point-à-point vers les modules adjacents. Le CPU présent dans chaque tuile est un processeur MIPS 32 bits avec, entre autres, une unité de calcul à point flottant. À haut niveau, les interconnexions permettent le déplacement de mots simples entre les unités connexes. Comme il a déjà été cité, les transferts de données sont décidés au moment de la compilation. Les auteurs indiquent toutefois qu'une solution dynamique est disponible pour un ordonnancement statique impossible, mais cette solution ne procure pas de bonnes performances et ne doit servir que si nécessaire.

Une synchronisation par les données n'est évidemment pas de mise pour une telle architecture, bien que plusieurs algorithmes de traitement de signal en bénéficieraient. Également, le focus ici est d'avoir des tuiles les plus petites possibles pour en mettre un maximum par puce pour augmenter le parallélisme et la vitesse d'horloge. Ceci a cependant pour effet une

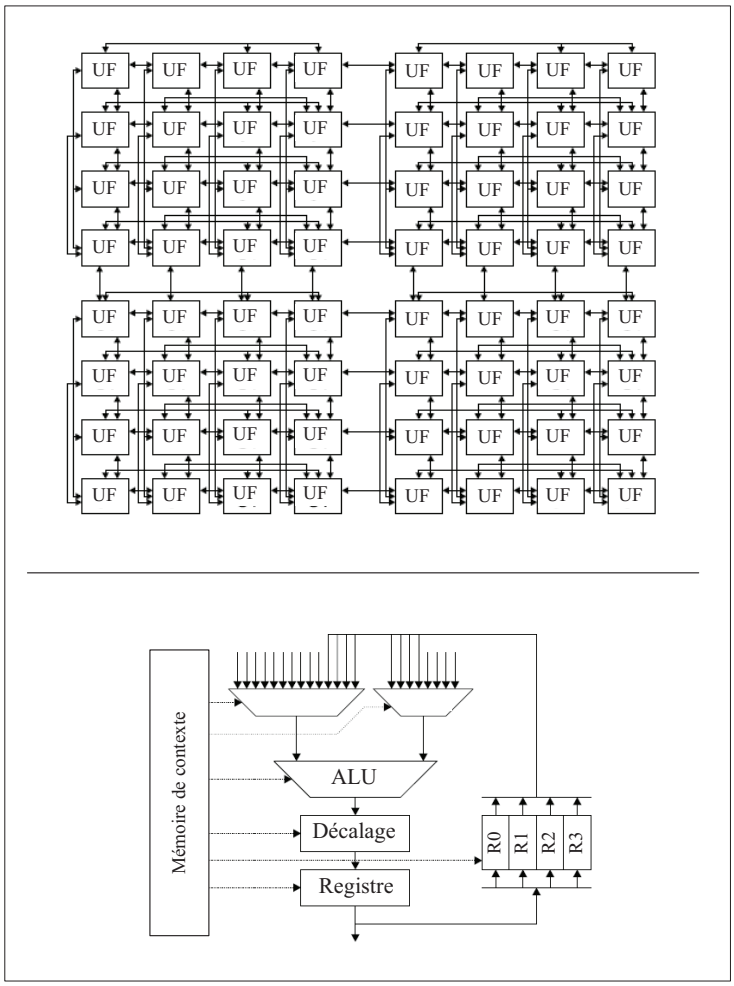


Figure 2.12 Treillis du MorphoSys (haut) et son unité fonctionnelle (bas)

consommation de ressources de 30% pour les interconnexions uniquement, ce qui n'est pas négligeable. Il n'y a également pas de logique configurable pour faire des instructions spécialisées, bien que l'idée fut présente au départ du projet. À leur défense, les RAM distribuées dans chaque unité de traitement permettent d'éliminer le problème de bande passante aux mémoires et offrent de meilleures latences individuelles (voir Waingold *et al.*, 1997).

### 2.5.4 MorphoSys

MorphoSys (*Morphoing System*) est un système de calcul comprenant un processeur standard étroitement relié à un coprocesseur reconfigurable. Bien que ce modèle de NoC puisse agir en temps que processeur général, il est dédié pour les applications ayant un grand niveau de parallélisme et une granularité élevée. Le design est optimisé pour des opérations sur des mots, mais est assez flexible pour supporter du traitement sur des bits. Plus spécifiquement,

MorphoSys est constitué d'un processeur MIPS, d'un contrôleur DMA, d'une mémoire de contexte et d'un treillis 8X8 de cellules reconfigurables.

Ce treillis ainsi qu'un exemple d'unité fonctionnelle sont illustrés en figure 2.12. La division en quadrant permet de voir les connexions : chaque ligne ou colonne peut être reconfigurée individuellement par le même mot de configuration. Chacune des unités fonctionnelles est composée de deux multiplexeurs à l'entrée, d'une ALU sur 16 bits, d'un registre à décalage , de 4 registres et d'une mémoire de contexte (pour la configuration). Vu sa nature SIMD, le MorphoSys est recommandé pour les applications multimédia traitant de hauts débits de données. Les auteurs indiquent toutefois qu'il faut remodeler le treillis si l'on désire avoir de meilleures performances selon les catégories d'applications. Finalement, le MorphoSys n'est pas conçu pour une implémentation sur FPGA (voir Singh *et al.*, 2000).

### 2.5.5 Architectures actuelles

Malgré un regain d'intérêt pour ce type d'architecture dans les années 1990, aucune d'entre elles ne vint définitivement révolutionner le monde des NoC reconfigurables, ce qui mit fin à la "ruée vers l'or". Il semble en effet que le type d'applications à accélérer ait beaucoup de poids dans la balance. Néanmoins, les recherches dans ce domaine n'ont pas cessées complètement et se ressemblent beaucoup puisque l'idée en soit a murie depuis le temps. À titre d'exemple, l'architecture développée par Hosseinabady (voir Hosseinabady et Nunez-Yanez, 2008), basée sur le schéma de la figure 2.7, permet de rouler les applications les plus fréquentes plus rapidement que les autres. Pour ce faire, le treillis est capable de faire circuler les applications de tuile en tuile afin d'offrir de meilleures latences lorsque désiré. On tente également d'éradiquer le processeur du système afin d'améliorer les performances. Cet ouvrage en est un parmi tant d'autres où l'architecture comme telle est classique mais implémente de nouvelles méthodes pour ordonnancer les tâches.

Bien entendu, certaines recherches poussent plus loin l'étude au niveau matériel, comme c'est le cas du EGRA (*Expression-Grain Reconfigurable Architecture*) qui consiste en une plateforme d'exploration de différents designs de CGRA (voir Ansaloni *et al.*, 2011). Sa structure est un treillis d'ALU regroupées (*cluster*), de mémoires et de multiplieurs. Sur la figure 2.13, nous pouvons voir un exemple d'un treillis 5X5. Il faut tout simplement faire le bon placement des différents modules (la grille n'est pas fixe). La symétrie diagonale, dans ce cas-ci, peut permettre deux flots séparés. On tente donc de trouver une façon de faire du placement automatique et même d'avoir une topologie irrégulière si les bénéfices sont présents. Pour l'instant, leur technique est réservée pour des experts dans le domaine.

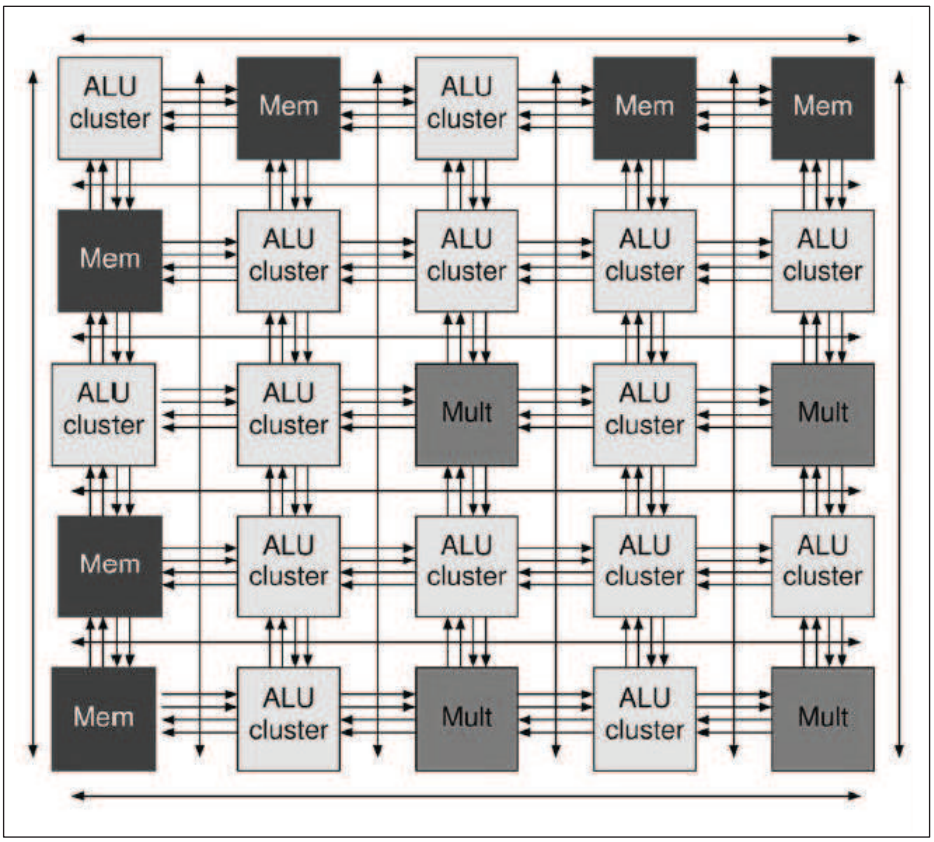


Figure 2.13 Architecture EGRA 5X5

Tableau 2.1 Caractéristiques d'architectures reconnues

| Nom | Disposition des UF | Reconfiguration | Chemin de données |
|---|---|---|---|
| PADDI | *Crossbar* | Statique | 16 bits |
| PADDI-2 | *Crossbar* | Statique | 16 bits |
| DP-FPGA | Treillis | Statique | 4 bits |
| KressArray | Treillis | Statique | 32 bits |
| Colt | Treillis | Dynamique | 16 bits |
| RaPiD | Linéaire | Statique | 16 bits |
| MATRIX | Treillis | Dynamique | 16 bits |
| Garp | Treillis | Statique | 2 bits |
| Raw | Treillis | Statique | 32 bits |
| PipeRench | Linéaire | Dynamique | 4 bits |
| REMARC | Treillis | Statique | 16 bits |
| MorphoSys | Treillis | Dynamique | 16 bits |
| CHESS | Treillis | Statique | 4 bits |

Une tendance très populaire dans le domaine des NoC est la reconfiguration dynamique. L'idée est de pouvoir modifier le comportement de l'architecture sans avoir à recharger le FPGA. On pourrait donc, par exemple, calculer de nouveaux algorithmes sur le flot de données, ou bien faire passer le trafic autrepart. Plusieurs études existent à ce sujet, dont celle de Killian (voir Killian *et al.*, 2012). Leur treillis consiste en une multitude de routeurs qui permettent de faire circuler les données d'un module IP vers un autre. En plus de contourner les liens défectueux, l'architecture permet de rajouter de façon dynamique de nouveaux modules (IP) pour rajouter des fonctionnalités. Le trafic doit cependant être redirigé et il faut être prudent de ne pas venir bloquer l'entrée d'un module existant.

Ces derniers travaux résument bien la pensée actuelle au sein de la communauté scientifique au sujet des architectures reconfigurables sur NoC. Il faut fournir un niveau matériel et un niveau logiciel assez flexible pour permettre une bonne collaboration tout en ayant une base ferme et performante pour ne pas empêtrer l'utilisateur. De plus, la reconfiguration dynamique sur FPGA, bien qu'étant un phénomène très récent et toujours en développement, pourrait offrir beaucoup d'avantages aux NoC au niveau de l'efficacité. En conclusion, le tableau 2.1 présente certaines caractéristiques pertinentes d'architectures reconnues qui n'ont pas nécessairement été citées dans le présent chapitre.

## 2.6 Conclusion

Dans ce chapitre, une revue de littérature sur les sujets propres aux architectures reprogrammables sur FPGA a été présentée. En premier lieu, il fut prouvé qu'une granularité fine offre un grand potentiel de parallélisme, mais au coût d'une synchronisation et d'interconnexions ardues. Les CGRA tentent donc de trouver un milieu intéressant entre les architectures à grains fins et à grains grossiers. D'autre part, les architectures à flot de données eurent un regain de popularité dans les années 1990 provenant de l'engouement vers les hybrides avec les travaux de Von Neumann. Plusieurs architectures, dont celles qui ont été présentées, exploitèrent cette particularité dans le but d'obtenir un NoC permettant d'accélérer des algorithmes dans tous les domaines (principalement en traitement de signal). L'élaboration de notre solution provient de l'étude de ces différentes recherches en la matière.

# CHAPITRE 3

# SOLUTION PROPOSÉE

## 3.1 Introduction

Comme nous l'avons vu au chapitre précédent, il devient rapidement complexe de concevoir une architecture de type treillis qui résout les principales difficultés de ce type de design (latence, coût en terme de ressources). Néanmoins, l'architecture réalisée et décrite dans le présent chapitre provient d'une étude approfondie des principaux enjeux ainsi que des capacités de la technologie FPGA. L'approche proposée est donc un treillis de calcul reconfigurable à deux niveaux d'abstraction. Au plus bas niveau (niveau matériel), le treillis est constitué de divers blocs qui réalisent la partie "contrôle" et la partie "donnée" du design. Au plus haut niveau (niveau logiciel), il est possible de configurer le treillis à travers de petites mémoires, permettant ainsi d'implémenter des algorithmes de façon logicielle en ayant des performances propres au matériel. La solution se veut une architecture à flot de données, dont celles-ci circulent par l'entremise de jetons (synchronisation par les données).

L'idée originale de l'architecture provient du professeur Jean Pierre David et de son analyse sur l'évolution des FPGA et des différents besoins pour faire du traitement numérique. L'architecture complète fut tout d'abord réalisée à l'aide du langage *SystemC* qui permet de rapidement modéliser une description matérielle, tant au niveau de l'architecture qu'au niveau des transactions entre les différents composants. C'est de cette façon que la plupart des protocoles de communication et le fonctionnement furent régis. En effet, il est plus simple et surtout plus rapide de corriger les anomalies par une méthode logicielle. Cette approche permet une exploration architecturale en imitant un langage de description matérielle mais à un niveau supérieur (*Transaction Level Modeling*). L'architecture finale provient du résultat de cette analyse. Ainsi, trois modules distincts sont utilisés pour concevoir le tissus de calcul.

La description du design complet se retrouve dans le présent chapitre. Tout d'abord, l'architecture à haut-niveau sera présentée pour introduire les modules composant le treillis : des *Machines à états*, des *Banques de FIFO* ainsi que des *SALU* (*Shared ALU*). Les principaux défis architecturaux ainsi que leur solution viendront prouver les forces du treillis. Également, un exemple de transaction à travers le treillis permettra de comprendre les bienfaits de la synchronisation par les données.

## 3.2 Architecture haut-niveau

En premier lieu, il est important de bien comprendre pourquoi on fait un treillis qui va gérer le chemin de contrôle et de données. Pour ce faire, utilisons l'exemple de la figure 3.1 où 4 FIFO sont connectées à 3 contrôleurs pour 3 ALU. Les entrées se dirigent vers ces FIFO, qui vont à leur tour servir à fournir les opérandes nécessaires aux opérations décidées par les contrôleurs. Les résultats sont ensuite disponibles pour devenir de nouveaux opérandes, ou bien pour sortir de l'architecture. Dans le cas idéal, chacune des FIFO est connectée à toutes les ALU présentes, comme nous pouvons le voir sur l'image. Le contrôleur doit donc gérer tout ce trafic et prendre les décisions. Imaginons maintenant un circuit avec des milliers de FIFO et des milliers d'ALU. En faisant toutes les connexions possibles, on aurait un contrôle qui serait quasi-impossible à implémenter. De plus, nous n'avons pas encore de processeurs ou de mémoires qui se rattachent au design. On voit rapidement qu'une topologie fixe qui limite les accès aux ressources permettrait d'optimiser l'implémentation d'algorithmes. Il faut donc créer un juste milieu entre le chemin de données et le chemin de contrôle.

De manière générale, le treillis de calcul se compose de plusieurs modules branchés entre eux et permettant de faire circuler des données par l'entremise d'opérations arithmétiques et logiques. La figure 3.2 montre un treillis de grandeur 5X5. L'architecture est composée

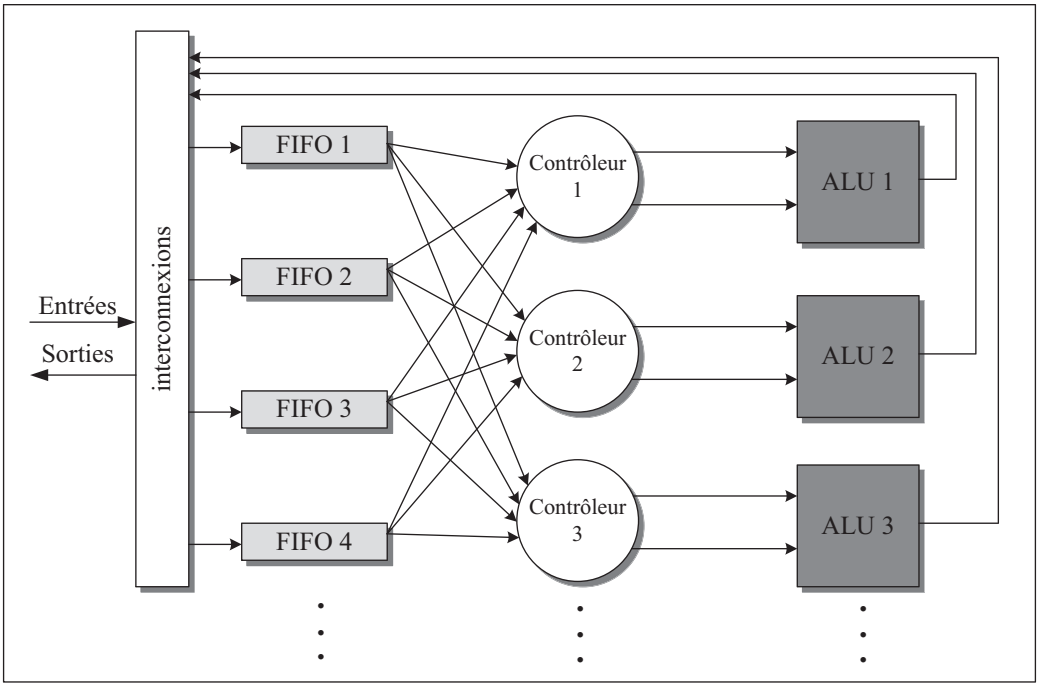


Figure 3.1 Exemple d'un système à 4 FIFO et 3 ALU

de 3 tuiles différentes connectées entre elles pour permettre de faire du traitement de signal. Malgré le fait qu'il faille conserver cet agencement pour s'assurer du bon fonctionnement, il serait facilement possible de modifier les interfaces pour inter-changer les blocs ou pour en créer de nouveaux (mémoires, interfaces *I/O*, processeurs, etc.). Également, le tout est extensible dans les 2 dimensions, selon les besoins de l'application que l'on désire réaliser. Le style "treillis" permet une grande flexibilité au niveau de l'exploration architecturale. Les performances locales ne sont pas dégradées par un agrandissement de la structure et les décisions de routage sont distribuées sur l'ensemble du modèle (en comparaison avec un bus). Pour une application donnée, les *Machines à états* fournissent des jetons d'instruction pour contrôler les transferts de jetons de donnée entre les *SALU* et les *Banques de FIFO*.

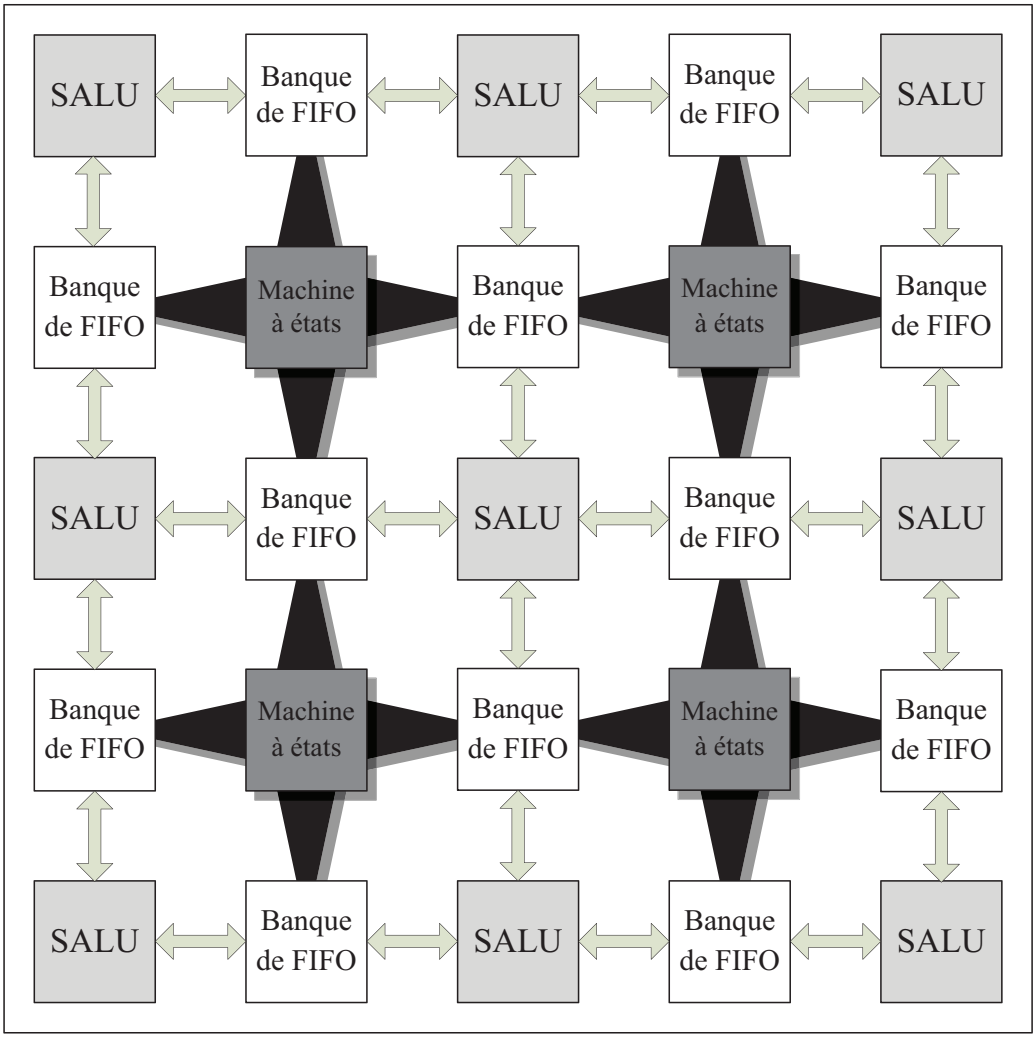


Figure 3.2 Treillis haut-niveau de dimension 5X5

Les *Machines à états* possèdent 8 petites mémoires indépendamment contrôlées renfermant les instructions de l'application. Celles-ci contiennent principalement les adresses des FIFO de source et de destination, ainsi que l'opération à effectuer comme telle. Ce module est donc responsable d'adresser les *Banques de FIFO* afin d'envoyer les bonnes données vers les *SALU*. Les instructions se suivent tant qu'il n'y a pas un blocage provenant des autres modules.

Les modules *Banque de FIFO* représentent les mémoires de l'architecture. Ils servent à stocker les résultats intermédiaires du tissu, les données d'entrée aux ALU ainsi que les entrées/sorties du système. Ils sont composés de FIFO individuellement adressables, similaire à une banque de registres, permettant un accès aléatoire (la donnée et/ou une copie de la donnée sont disponibles). Chacune des banques est contrôlée par 4 mémoires d'instructions (2 par chaque voisin horizontal ou vertical). Lorsqu'une nouvelle instruction est disponible, le module envoie les opérandes vers les bonnes *SALU*, et gère les retours de résultats des différentes opérations. Les *Banques de FIFO* permettent de faire le transport des jetons de donnée à travers l'architecture.

Le dernier module, conçu pour faire les différentes opérations arithmétiques et logiques, est la *SALU*. Il contient 8 petites ALU accessibles par les *Banques de FIFO* (chaque paire est contrôlée par une banque différente). Les ALU peuvent effectuer des tâches simples tel un décalage, une addition/soustraction, ou des tâches plus complexes comme une division ou même une racine carrée. Afin de réaliser le transport entre toutes les opérations effectuées et les FIFO de résultats externes au module, un petit réseau de routeur inscrit dans la *SALU* permet de router les jetons. Une particularité intéressante du design est qu'il est possible pour l'utilisateur de venir décrire à bas niveau ses propres opérations et d'y avoir accès par un simple appel d'instruction.

Un aspect très important de cette architecture est qu'elle fonctionne selon le principe des jetons. Ceux-ci se promènent donc de FIFO en FIFO, à travers les *SALU*. La décision architecturale d'utiliser des FIFO permet d'aisément contrôler cette synchronisation par les données. De plus, le treillis complet est en pipeline et le flot de données reste constant. Dans le cas éventuel où une donnée se fait attendre, le pipeline se remplit et se bloque pour reprendre dès que possible ; le tout se déroule sans avoir un système complexe pour le contrôle.

Un avantage majeur de l'architecture proposée est son évolution adaptative en fonction de la technologie FPGA qui évolue elle aussi. En effet, les nouvelles familles de FPGA qui font surface sur le marché offrent toujours de nouvelles particularités en comparaison aux itérations précédentes. On se retrouve ainsi avec des interfaces améliorées, de nouvelles mémoires

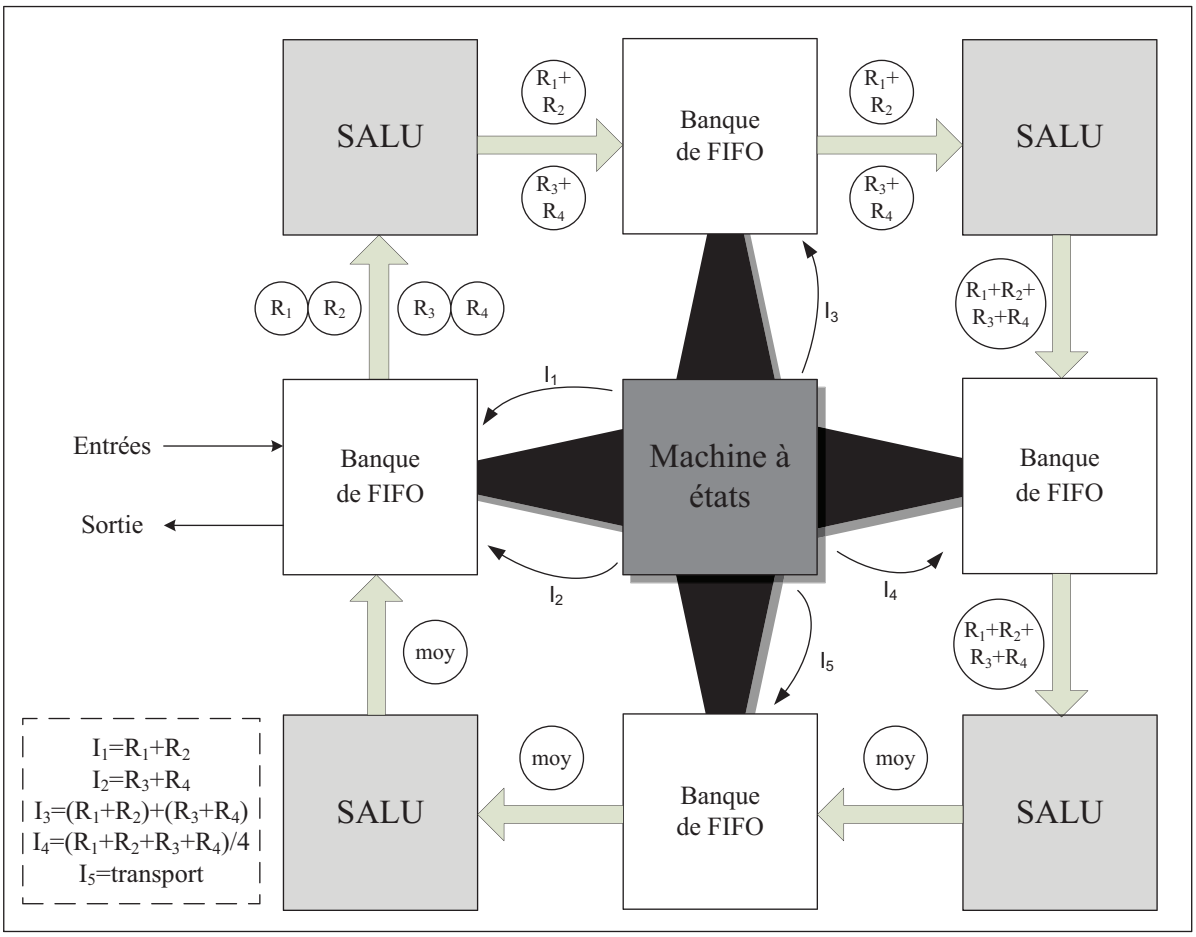


Figure 3.3 Exemple du calcul de la moyenne d'un flot de 4 valeurs sur le treillis

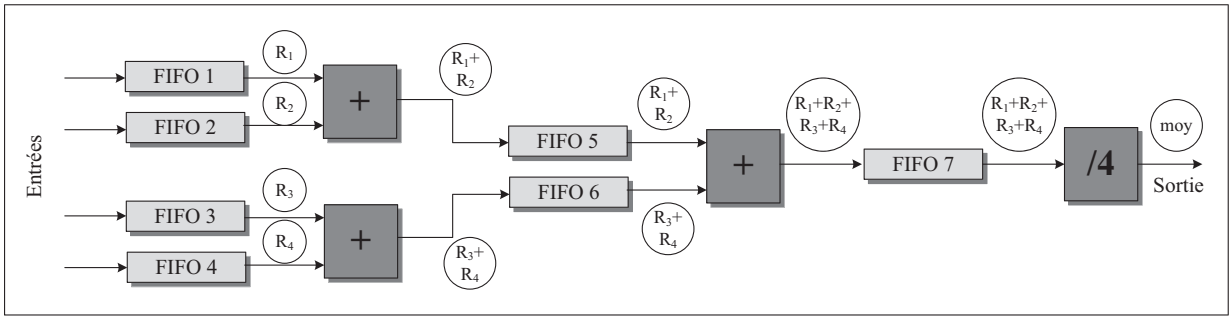


Figure 3.4 Exemple du calcul de la moyenne d'un flot de 4 valeurs sans le treillis

distribuées et de plus en plus de ressources. En prenant en exemple l'architecture RaPiD sur du matériel dédié, on réalise que son évolution nécessite beaucoup d'effort. La raison pour laquelle aucune nouvelle mouture n'a eu lieu depuis sa création provient en grande partie du fait qu'il faut tout repenser en fonction du progrès technologique des ASIC. Notre solu-

tion proposée, quant à elle, s'ajuste aux nouvelles avancées et ne sera pratiquement jamais désuette.

L'exemple illustré sur la figure 3.3 démontre le calcul de la moyenne de 4 registres (représentant les valeurs de sorties des FIFO). Son pendant en dehors du treillis se retrouve en figure 3.4 pour des fins de comparaison et de compréhension. Les instructions $I_1$ et $I_2$ indiquent à la *Banque de FIFO* située à la gauche d'envoyer respectivement $R_1$,$R_2$ et $R_3$,$R_4$ vers la SALU pour faire des additions. Ces 4 jetons sont donc consommés pour former des jetons de résultats, envoyés en mémoire dans la *Banque de FIFO* supérieure. De la même manière, l'instruction $I_3$ permet de faire l'addition finale tel qu'indiqué sur la figure. Par la suite, l'instruction $I_4$ effectue la division par 4 pour obtenir la moyenne, et l'instruction $I_5$ ne fait que propager le jeton final vers la case initiale. Il est à noter qu'à chaque cycle, de nouvelles transactions ont lieu et que les jetons $R_1$ à $R_4$ prennent de nouvelles valeurs. Également, il est possible de réaliser la moyenne avec un plus petit nombre de tuiles ; tout dépend de ce qu'on désire accomplir en terme de performance et de coût.

Une telle architecture offre ainsi à des utilisateurs peu expérimentés dans le domaine de la description matérielle de programmer de façon logicielle leurs applications, et d'en retirer les avantages du parallélisme. Il suffit de programmer les *Machines à états* et le reste du circuit est transparent. Les possibilités sont énormes puisque plusieurs algorithmes de nature dépendante ou non peuvent fonctionner à travers le treillis sans toutefois interférer entre eux. Il faut cependant avoir une bonne compréhension du design afin de correctement utiliser les mémoires d'instruction.

### 3.3 Module "Machine à états"

Le premier module est celui qui représente l'interface directe avec l'utilisateur du treillis. Le but premier des *Machines à états* est de permettre de la manière la plus simple possible l'implémentation d'algorithmes de traitement de signal. Il faut donc pouvoir générer plusieurs instructions en même temps pour fournir toutes les *Banques de FIFO* voisines. Ces instructions doivent contenir toute l'information nécessaire pour choisir les opérandes, l'opération, le lieu d'opération ainsi que la destination. Le schéma de la figure 3.5 étaye la solution proposée.

Tout d'abord, le module *Machine à états* contient en fait 8 petites machines, dont chacune possède sa propre interface indépendante. Chaque paire correspond à une *Banque de FIFO* différente (voir figure 3.2). Initialement, il n'y avait que 4 machines, mais une exploration

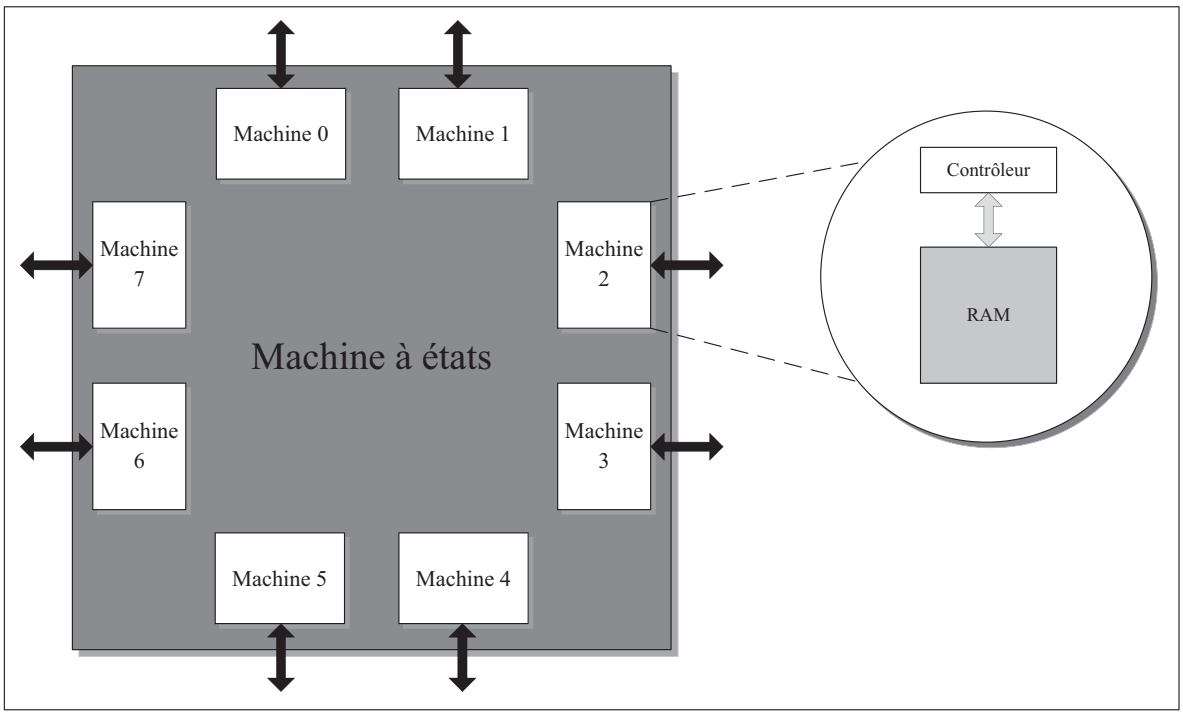


Figure 3.5 Machine à états

architecturale en fonction de différents algorithmes démontra qu'on pouvait augmenter l'exploitation du parallélisme de pratiquement tous les algorithmes de cette manière. On peut comprendre ce concept en remarquant que chaque *Banque* est entourée de 2 $SALU$, et qu'il devient ainsi plus simple de contrôler les jetons de données. Il est toujours possible d'en hausser le nombre si nécessaire, mais on augmente également les sous-modules des autres tuiles, ce qui finit par complexifier les implémentations d'algorithmes.

Puisque chacune de ces machines est indépendante, on doit pouvoir les programmer de façon indépendante. C'est pourquoi chaque petite machine comporte sa propre mémoire RAM dédiée ainsi qu'un contrôleur. Les mémoires contiennent la totalité des instructions nécessaires afin d'exécuter l'algorithme désiré par l'utilisateur. C'est cette fonctionnalité qui assurera au treillis d'être reconfigurable pré-synthèse et post-synthèse. Chacun des contrôleurs est utilisé pour accéder aux instructions et faire les envois nécessaires qui permettront d'adresser les *Banques de FIFO* et d'utiliser les bons jetons de donnée. De plus, ils représentent les machines à états en ce sens qu'ils prennent les décisions sur les prochaines instructions à réaliser en fonction des retours (par exemple, un contrôleur décidera de sauter certaines instructions ou non en fonction du drapeau d'état "négatif"). Si la condition est respectée, l'instruction est exécutée. Dans le cas contraire, la machine à états passera à l'instruction suivante. Cela

Tableau 3.1 Mot d'instruction dans les machines à états

| Bits | Description |
|---|---|
| 46-43 | Opération |
| 42-39 | Condition |
| 38 | Sélection de la *SALU* |
| 37 | Modification des drapeaux |
| 36 | Provenance de l'opérande A (*Banque* ou *SALU*) |
| 35 | Lecture profonde de l'opérande A |
| 34-30 | Adresse de l'opérande A |
| 29-26 | Adresse de lecture profonde de l'opérande A |
| 25 | Provenance de l'opérande B (*Banque* ou *SALU*) |
| 24 | Lecture profonde de l'opérande B |
| 23-19 | Adresse de l'opérande B |
| 18-15 | Adresse de lecture profonde de l'opérande B |
| 14-12 | *Banque* ou ALU de destination |
| 11 | Écriture en FIFO |
| 10 | Adresse d'écriture double ou choix du tampon |
| 9-5 | Adresse 1 du résultat ou vide |
| 4-0 | Adresse 2 du résultat ou vide |

permettra aussi d'effectuer des branchements qui donneront la possibilité d'implémenter des conditions et des boucles. On peut voir chaque machine comme un petit processeur avec des instructions matérielles spécialisées (*SALU*). Une machine sera en état d'arrêt tant que l'opération qu'elle vient d'envoyer n'aura pas été effectuée. Cela veut donc dire que si une partie d'une *SALU* est bloquée, toute la machine est bloquée jusqu'à ce que la *SALU* fautive se libère. Il faut donc bien réfléchir au sujet des implémentations algorithmiques pour éviter des problèmes de *deadlock*. Le tableau 3.1 décrit en détail les mots d'instruction sur 47 bits.

Quand on conçoit une instruction, l'utilisateur doit tout d'abord définir l'opération et la condition. Le premier champ désigne une opération parmi toutes celles disponibles implémentées dans les *SALU*. On peut également choisir de faire un vidage d'une FIFO ou d'ordonner un branchement. Celui-ci aura lieu si la condition est respectée (résultat plus petit ou plus grand qu'une certaine valeur, débordement signé, etc.). On définit par la suite la *SALU* désirée et si la modification des drapeaux d'état est nécessaire. Ces drapeaux sont :

1. Z : Si Z est à 1, le résultat de la dernière opération affectant les drapeaux est 0 ;
2. N : Si N est à 1, le résultat de la dernière opération est négatif ;
3. C : Si C est à 1, le résultat de la dernière opération engendre un débordement ;

4. V : Si V est à 1, le résultat de la dernière opération engendre un débordement (signé).

Par la suite, le programmeur indique les sources des 2 opérandes. Afin d'assurer une grande flexibilité, ceux-ci peuvent se retrouver à plusieurs endroits mais sont toujours disponibles à partir de la *Machine à états*. On définit donc la source comme étant un des tampons internes d'une *SALU* voisine ou une FIFO provenant de la *Banque de FIFO*. Dans ce dernier cas, il faut spécifier l'adresse de la FIFO et de la lecture aléatoire, si nécessaire. Au niveau de la destination, on doit tout d'abord choisir vers quelle côté de la *SALU* le résultat devra être envoyé. Un bit est utilisé pour déterminer si le résultat doit être écrit en mémoire (*Banque de FIFO*) ou s'il sera repris par une ALU de destination (*SALU*). Dans le premier cas, les bits restants servent à identifier les adresses des résultats (possibilité d'écriture double si désiré). Pour une écriture en *SALU*, on choisit le tampon adéquat (nous verrons ces tampons dans la section de la *SALU*).

La grandeur actuelle d'une instruction du treillis (47 bits) provient de différents essais sur le modèle *SystemC*. La dimension des champs dépend du nombre de FIFO, de leur grandeur, du nombre d'opérations réalisables en *SALU*, etc. Au stade actuel, certaines portions de l'*opcode* sont plus grandes pour permettre au programmeur de rajouter de la mémoire ou des opérations si nécessaire.

En dernier lieu, il est important d'examiner l'interface entre la *Machine à états* et les *Banques de FIFO* environnantes. Puisque chacune des machines ne dépend que d'elle-même, elles auront toutes leur propre entrées/sorties, mais qui seront identiques. Outre les signaux d'usage comme l'horloge et le signal de réinitialisation, on doit bien entendu avoir comme sortie l'instruction. Celle-ci est sur 42 bits (les autres ne servent qu'à la machine). Le retour des drapeaux d'état provenant des *SALU* est également très important pour permettre les branchements. La dernière paire d'entrée/sortie est le RTS (*Ready To Send*) et le RTR (*Ready To Receive*). Un défi rencontré lors de l'élaboration de l'architecture était de s'assurer de ne pas envoyer d'instruction si le flot de calcul est bloqué. Dans le cas présent, on doit s'assurer que les jetons de données soient présents et que la FIFO accueillant le résultat ne soit pas pleine. La description matérielle pour réaliser le RTR sera décrite dans une prochaine section. Pour ce qui est de la sortie RTS, il suffit de regarder si la prochaine instruction est disponible et si l'on désire faire une opération par une *Banque de FIFO* et/ou une *SALU*.

En reprenant l'exemple de la figure 3.3, les machines actives (1,3,4,6,7) ne contiennent en fait qu'une mémoire RAM d'un seul mot d'instruction. Les instruction $I_1$ à $I_3$ produisent la même addition à chaque coup d'horloge, mais avec des opérandes différents. Il en va de même

pour $I_4$ (division) et $I_5$ (propagation). En fonction de l'*opcode* présenté précédemment, voici l'instruction $I_1$ :

– $I_1$="0001|0000|0|0|1|0|00001|0000|1|0|00010|0000|010|1|0|00001|00000".

On opère donc une addition sans condition et sans modification des drapeaux, sur la *SALU* supérieure. Pour l'opérande A, on choisit la FIFO 1 sans lecture profonde, et la FIFO 2 sans lecture profonde pour l'opérande B. Le résultat ira vers la droite de la *SALU*, en *Banque de FIFO* sur la FIFO 1. Les prochaines sections détailleront de quelle manière le résultat final s'obtient à partir d'une instruction.

### 3.4 Module "Banque de FIFO"

Le module *Banque de FIFO* représente de façon générale la mémoire du système. En analysant le schéma haut-niveau du treillis (figure 3.2), il est clair que c'est le bloc le plus présent dans l'architecture, et le seul qui a une connexion avec 2 entités différentes. En effet, son objectif principal est de faire la liaison entre les *Machines à états* et les *SALU*. De cette façon, on allège les machines (et ainsi le travail du programmeur) en ayant de la mémoire de données de façon uniforme. On peut voir le module comme étant le fil de communication et de transport à travers le tissu. Un important prérequis était de pouvoir faire des lectures et des écritures simultanées, tout en conservant l'idée du jeton de donnée. La synchronisation par les données est la fondation de l'architecture, et c'est principalement à ce niveau que ça se réalise. La *Banque de FIFO* doit également gérer tout le trafic nécessaire pour acheminer les jetons de donnée et les instructions aux *SALU* environnantes, tout en permettant le pipelinage réglé au cycle d'horloge. Le schéma de la figure 3.6 démontre les principales caractéristiques de cet élément clé du tissu de calcul.

Notons en premier lieu que cette *Banque* est entourée de *SALU* verticalement et de *Machines à états* horizontalement, mais qu'il suffit de faire une rotation de 90° pour obtenir le deuxième type. L'élément de mémoire qui fut sélectionné est la FIFO. En effet, pour permettre la synchronisation par les données, il est primordial d'avoir la capacité de connaître l'ordre d'arrivée des jetons. La sortie de chacune des FIFO agit comme un registre, d'où la numérotation $R_0$ à $R_{15}$ (le nombre de FIFO et leur grandeur sont paramétrables avant-synthèse). Prenons l'exemple de la figure 3.7 où l'on désire additionner $R_1$ et $R_2$, Si la première FIFO dénote une latence initiale de deux cycles avant d'obtenir son premier jeton, on aura une accumulation de deux jetons dans la deuxième FIFO à chaque cycle d'horloge subséquent, mais on s'assurera de respecter l'ordre d'arrivée pour opérer les additions.

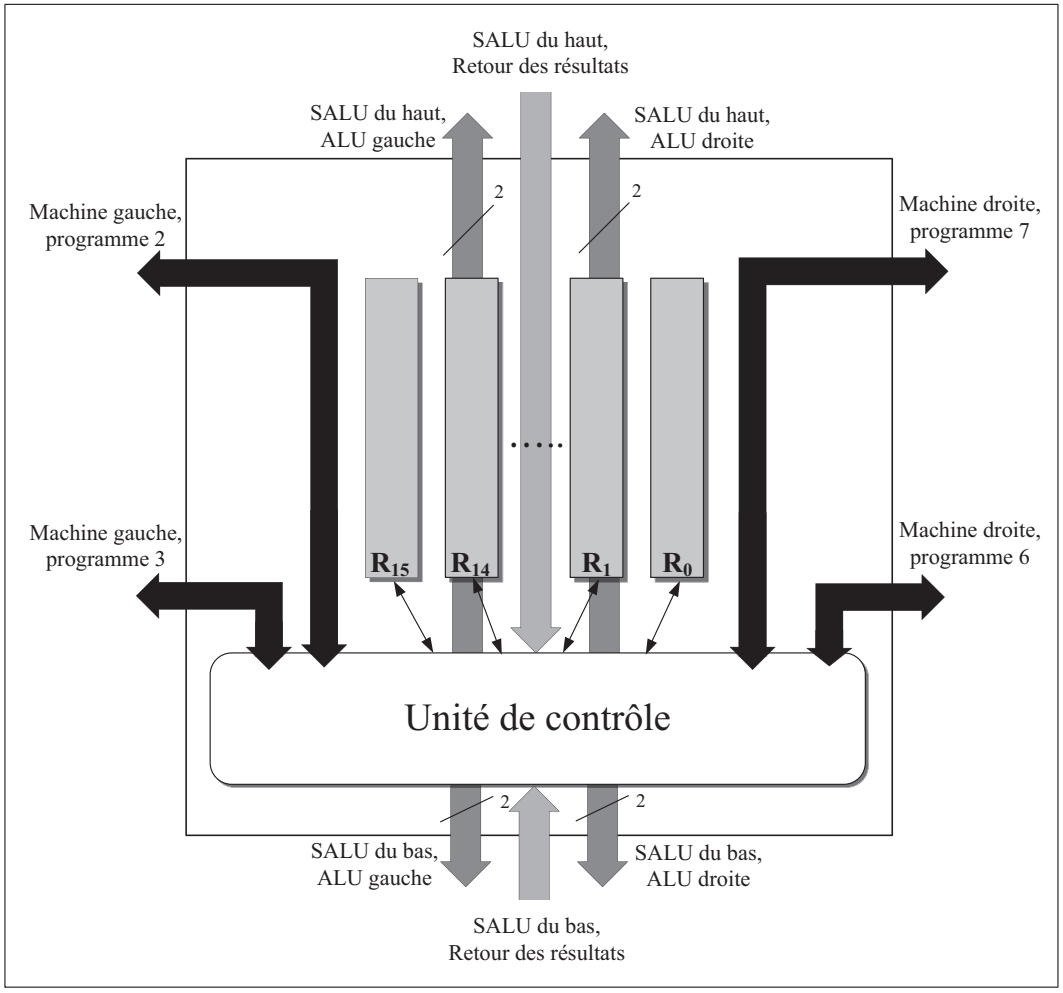


Figure 3.6 Banque de FIFO

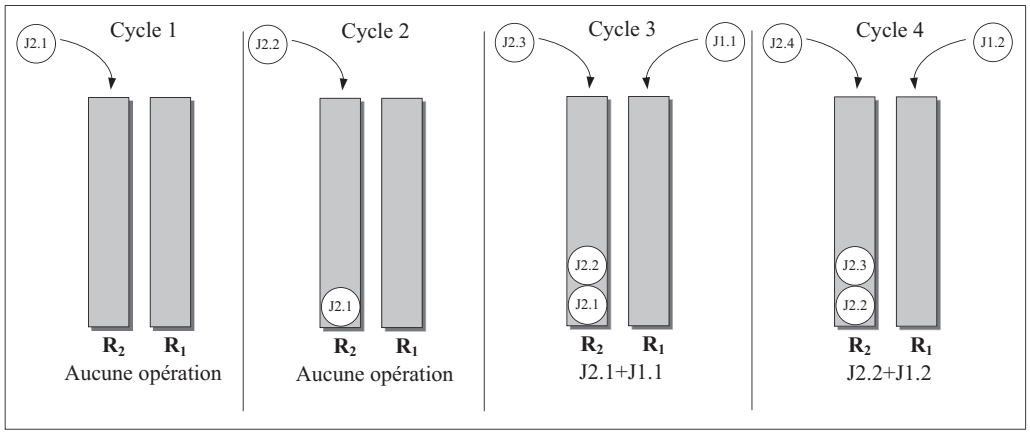


Figure 3.7 Exemple d'addition de $R_1$ et $R_2$

Cette série de FIFO adressables individuellement, en plus de permettre la lecture et l'écriture simultanée, admet la lecture aléatoire (ou lecture profonde). En effet, il est parfois utile de seulement faire une copie de la donnée et de la conserver en mémoire, ce qui n'est pas le cas avec une FIFO usuelle. Il est donc possible d'indiquer l'adresse de celle dont nous voulons copier un jeton. Cette caractéristique est particulièrement utile pour utiliser des constantes à travers le tissu de calcul. En sus des caractéristiques de base (signal de lecture/écriture, indication *full*, *empty* et *size*), il est possible de faire un vidage d'une FIFO à tout moment par une simple instruction. On peut aussi initialiser les constantes dans ces mémoires RAM.

Outre les FIFO, le module fonctionne complètement en logique combinatoire. Les mémoires qui se retrouvent à travers le treillis, autant dans les *Banques* que dans les *SALU*, représente les étages du pipeline. L'unité de contrôle présente sur la figure 3.6 est responsable de la bonne gestion de tous les signaux entrants et sortants du module. En tout temps, cette unité regarde parallèlement l'envoi des RTS des 4 machines voisines. À l'arrivée d'une nouvelle instruction, un décodeur analyse les sources et la destination pour vérifier la disponibilité, et fait les envois nécessaires le cas échéant.

Chacun des opérandes peut provenir d'une FIFO présente dans la *Banque* (une copie ou non) ou d'un tampon provenant d'une *SALU*. Il y a donc 9 cas possibles. Lorsqu'on veut une copie d'un jeton, on doit s'assurer que l'adresse de la donnée est plus petite ou égale au nombre d'éléments de la FIFO en question. Si c'est le cas, l'unité de contrôle s'assure de faire un duplicata de cette valeur. Pour faire sortir un jeton d'un élément de mémoire, le signal *empty* est analysé, de sorte à rendre la donnée disponible. Il en va de même pour une provenance externe, c'est-à-dire que l'unité de contrôle a accès au signal de présence provenant des tampons dans la *SALU*. Il est important de noter que c'est uniquement lorsque les opérandes sont disponibles et que la *SALU* dépêche son propre signal de RTR que l'on envoit tous les signaux en indiquant par le fait même à la machine de passer à la prochaine instruction (elle reçoit un RTR de la *Banque*). Advenant le cas, par exemple, qu'un jeton est manquant, la machine ne pourra poursuivre son processus tant que la donnée n'est pas arrivée. On s'assure ainsi de ne sauter aucune étape de l'algorithme global. C'est également la raison pour laquelle le programmeur haut-niveau doit bien s'assurer de ne pas créer de *deadlock* dans le treillis.

Les 4 machines ayant accès à une *Banque de FIFO* donnée peuvent contrôler deux *SALU*. Cependant, la prochaine section fera la démonstration que chaque *SALU* contient 8 petites ALU pour être concordant avec le nombre de machines disponibles. L'unité de contrôle de

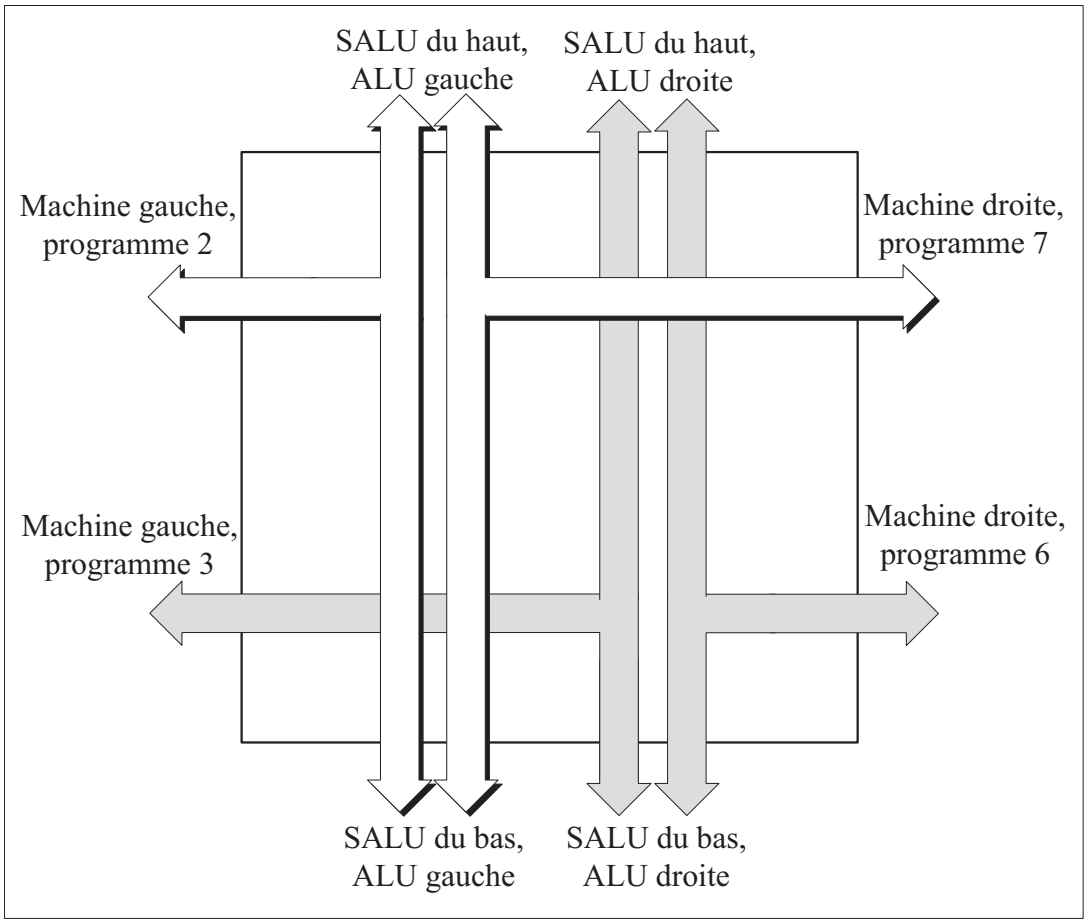


Figure 3.8 ALU disponibles pour chaque machine

la figure 3.6 gère 2 paires d'ALU ; supérieure et inférieure. Cette gestion signifie de faire le routage des opérandes, des instructions, des drapeaux d'états et de tous les signaux provenant des FIFO. Par la suite, les 2 *SALU* possèdent un canal pour faire le retour de jetons de type "résultat". Ceux-ci sont sauvegardés en mémoire (possibilité d'écriture double).

Afin d'assurer une grande souplesse au tissu de calcul, chacune des machines doit pouvoir choisir la *SALU* de son choix. Cependant, une certaine limitation s'impose sur l'ALU comme telle. En effet, l'architecture limite les conflits selon les possibilités de la figure 3.8. Les flèches ne représentent pas des liens physiques mais bien la possibilité de contrôle de chacune des machines (les liens *SALU-SALU* ne sont pas possibles). Cet agencement arbitraire concède à chaque machine la possibilité de choisir le sens du chemin de données.

À ce stade, un questionnement s'impose : de quelle façon la décision du routage entre les interfaces des *SALU* et des *Machines à états* à travers les *Banques de FIFO* doit se faire ?

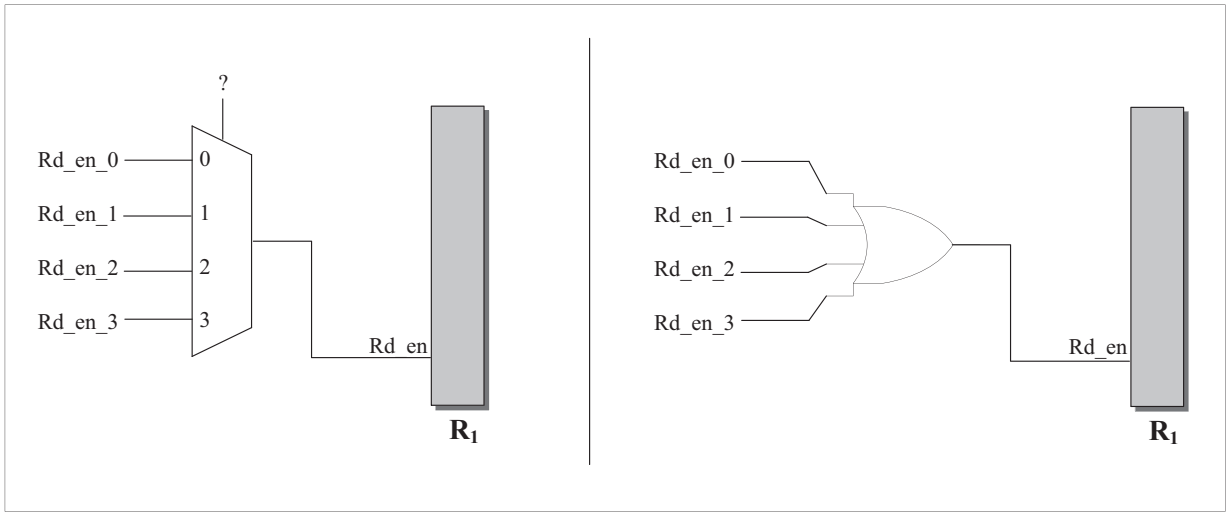


Figure 3.9 Exemple de sélection du *Rd_en*

En effet, la décision de permettre aux différentes ALU d'être contrôlées par plus d'une machine entraîne un problème de multiplexage. Prenons l'exemple où la machine de gauche (programme 2) désire utiliser la *SALU* supérieure, tout comme la machine de droite (programme 6). Les deux ALU supérieures seront utilisées sans conflit. Par contre, advenant le cas où la machine de droite (programme 7) tente d'accéder à la *SALU* supérieure, il y aura un conflit pour l'accès à la ressource. Dans le même ordre d'idée, qu'arrivera-t'il si deux machines veulent utiliser la même FIFO ? Il en va de même pour tous les signaux ainsi partagés.

Ce défi architectural est représenté par un exemple sur la figure 3.9 (côté gauche). On tente d'accéder à la FIFO $R_1$ par les 4 machines. Il faut donc trouver une manière de donner l'accès à une seule d'entre elles. On peut régler le problème en utilisant une technique de priorité tel que le *round-robin*, mais ça devient rapidement impossible à gérer car les jetons continuent d'affluer dans la *Banque*, et on ne peut plus garantir la bonne association données-instruction. La solution envisagée est d'obliger le programmeur à associer les ressources afin d'éviter ce problème à haut-niveau. Cependant, il faut tout de même régler le problème au niveau matériel et rendre les connexions possibles. Le côté droit de l'image 3.9 montre qu'avec une simple porte OU, on corrige le problème (il faut qu'un seul des signaux soit à '1'). Le coût en est également grandement réduit. Cette astuce est utilisée sur tous les signaux conflictuels (opérandes, drapeaux d'état, signaux de contrôle, etc.).

En résumé, le module *Banque de FIFO* sert de mémoire dans le tissu de calcul par l'entremise de FIFO conservant les jetons de donnée. Ces jetons circulent de module en module sous mode consommation-création à travers les *SALU* présentes sur le chemin. Chacune des

Tableau 3.2 Mot d'instruction vers les *SALU*

| Bits | Description |
|---|---|
| 21-18 | Opération |
| 17 | Modification des drapeaux |
| 16 | Provenance de l'opérande A (*Banque* ou *SALU*) |
| 15 | Provenance de l'opérande B (*Banque* ou *SALU*) |
| 14-12 | *Banque* ou ALU de destination |
| 11 | Écriture en FIFO |
| 10 | Adresse d'écriture double ou choix du tampon |
| 9-5 | Adresse 1 du résultat ou vide |
| 4-0 | Adresse 2 du résultat ou vide |

*Banques* fait donc la liaison entre les tuiles environnantes et gère tous les envois de manière purement combinatoire. Cette particularité est cruciale pour permettre aux liaisons RTS-RTR de fonctionner et d'avoir un pipeline donnant un résultat final à chaque cycle. En prenant un cycle supplémentaire à chaque instruction pour calculer le RTR, nous ne pourrions avoir un flot de données entrant régularisé sur chaque front montant d'horloge. Puisque la totalité des ressources pour ce bloc sert à faire du routage, la fréquence maximale du circuit ne devrait pas en être trop affectée. De plus, le mot d'instruction provenant des différentes machines est envoyé vers les *SALU* avec quelques retraits pour ne garder que le nécessaire. Le tableau 3.2 propose ce nouvel *opcode*.

## 3.5 Module "SALU"

Le dernier module présent dans le tissu de calcul proposé est la *SALU*. Tout comme un processeur à usage général, une unité arithmétique et logique est indispensable pour faire du traitement de données. Dans le cas présent, la solution devrait permettre en tout temps d'opérer jusqu'à 8 opérations simultanément et indépendamment les unes des autres. Afin d'assurer un flot de données constant et synchronisé à travers l'architecture, la *SALU* doit posséder un réseau d'interconnexions interne pour acheminer les jetons résultat soit vers les *Banques de FIFO* adjacentes, soit vers un des nombreux tampons internes. Le défi au niveau matériel est de faciliter les calculs et les communications des jetons qui afflueront à travers le module, sans causer de latence non-nécessaire. Du point de vue externe à la *SALU*, on doit pouvoir lui fournir jusqu'à 8 opérations par cycle et de garantir le bon cheminement des données. Les ALU doivent opérer des calculs simples et complexes tout en mettant à jour les différents drapeaux d'états. Le schéma de la figure 3.10 expose l'approche développée.

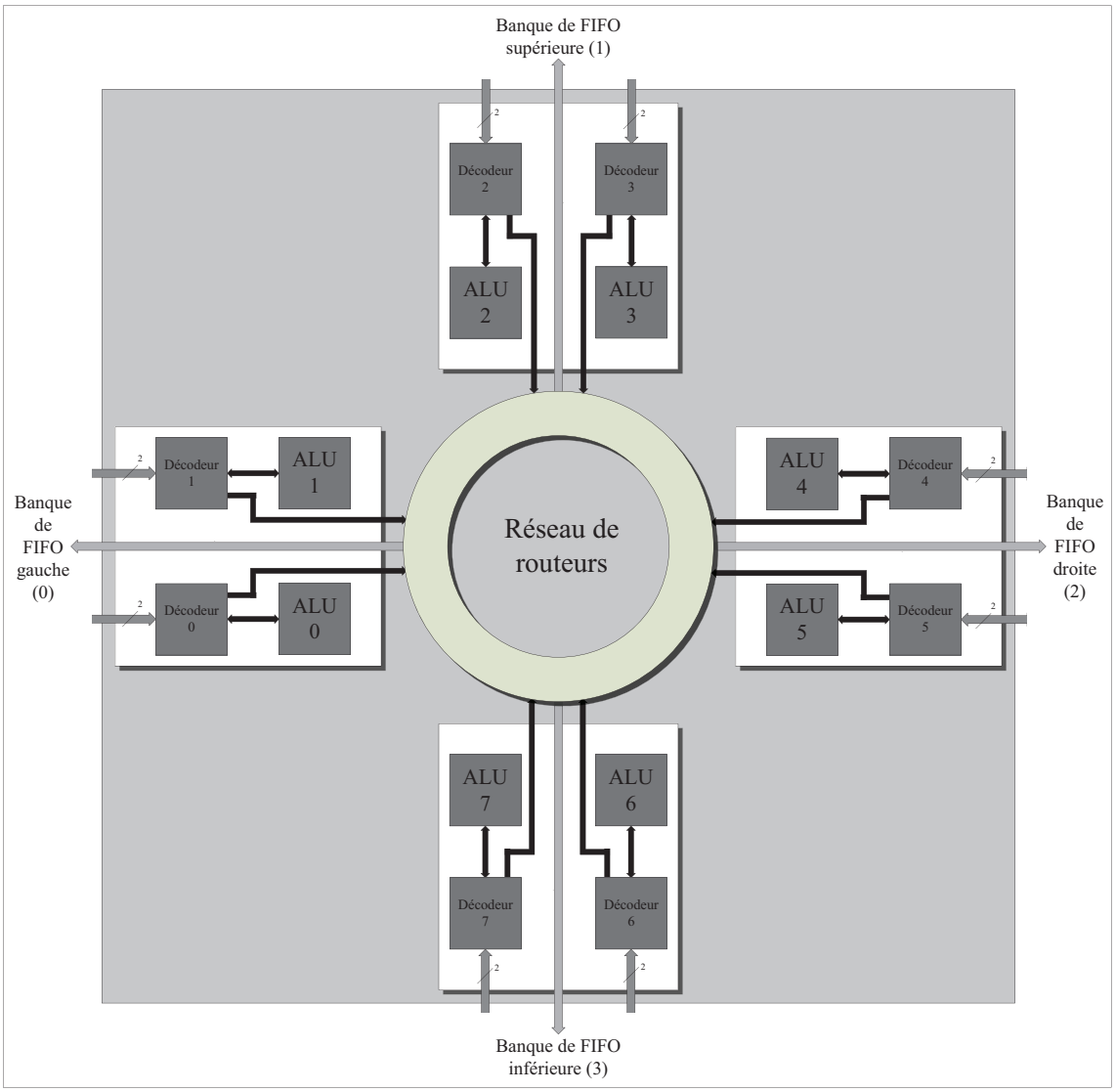


Figure 3.10 *SALU*

En sachant que ce module est entouré par 4 *Banques de FIFO*, les différentes flèches représentant les interfaces externes se retrouvent en figure 3.6. La *SALU* est composée de 8 petites ALU indépendantes contrôlées par leur propre décodeur. Chaque tuile voisine a donc accès à 2 unités arithmétiques et logiques, en plus d'un retour de résultats. Lorsqu'une opération est terminée, le jeton ainsi créé est acheminé vers sa destination par un réseau de routeurs. Les bienfaits de cette technique sont qu'un échange organisé et séquentiel est possible pour gérer tout le trafic de jetons. Le réseau doit cependant être conçu pour favoriser certains échanges et avoir un pipeline bien serré. Les numérotations sur la figure 3.10 sont indispensables pour l'adressage provenant d'aussi loin que les *Machines à états*.

### 3.5.1 Décodeur

Ce sous-module est utilisé pour faire le lien entre les envois d'une *Banque de FIFO* et l'ALU désirée, pour ultimement acheminer les jetons résultat à bon port, le tout de façon purement combinatoire. La première étape est d'acheminer les opérandes et l'opération à faire vers l'ALU. Comme nous l'avons vu précédemment, la *Banque* est responsable d'acheminer les opérandes vers le bloc décodeur. Cependant, l'ALU peut également effectuer des opérations sur des jetons provenant de tampons internes à la *SALU*. La décision d'instaurer ces petits éléments de mémoire provient d'une réflexion sur le flot de données à travers l'architecture. Prenons le cas où l'on désire faire de multiples soustractions d'un nombre par une constante. Cette constante sera irrémédiablement mise en *Banques* avant la synthèse, et le résultat serait sauvegardé en tampon interne pour éviter de bloquer une FIFO inutilement. Ces petites caches près des ALU sont implémentées comme des FIFO à 1 mot. Il est impossible pour le réseau de routeurs d'écraser un jeton déjà présent dans ces tampons ; le pipeline se bloque le cas échéant. Chaque décodeur possède deux de ces tampons pour satisfaire aux deux opérandes possibles. Il est important de noter ici que toutes les combinaisons sont admissibles, c'est-à-dire qu'on pourrait avoir un premier opérande provenant de la *Banque* adjacente et un deuxième provenant d'un tampon.

Le décodeur envoie ensuite les opérandes et les jetons vers l'ALU pour recevoir le résultat. Celui-ci sera par la suite acheminé vers sa destination finale à travers le réseau de routeurs. Il faut bien comprendre que toutes les décisions de routage se font à travers le réseau. Ainsi, même si un décodeur possède un résultat qui ira dans un de ses propres tampons, le jeton doit tout de même avoir été géré par le système de communication interne de la *SALU*. Les routeurs se partagent ainsi des mots contenant à la fois le contrôle et la donnée. Une version brouillon du tissu séparait ces deux instances mais ralentissait ainsi le flot de données complet. Ces mots de 64 bits, construits par le décodeur, sont détaillés dans le tableau 3.3.

Tableau 3.3 Mot de contrôle et de donnée vers le réseau de routeurs

| Bits | Description |
|---|---|
| 63 | Écriture en FIFO |
| 62-61 | *Banque* de destination |
| 60 | Adresse d'écriture double |
| 59-55 | Adresse 1 du résultat |
| 54-50 | Adresse 2 du résultat |
| 49-32 | Vide |
| 31-0 | Jeton de donnée |

| Bits | Description |
|---|---|
| 63 | Écriture en tampon interne |
| 62-60 | Décodeur de destination |
| 59 | Tampon gauche ou droit |
| 58-32 | Vide |
| 31-0 | Jeton de donnée |

Du côté gauche, nous avons le mot d'instruction permettant d'envoyer le jeton vers une FIFO présente dans une *Banque* (bit 63 à '1'), alors que le côté droit est utilisé pour un envoi en tampon interne (bit 63 à '0'). Dans les deux cas, les derniers 32 bits contiennent le jeton de résultat(la taille est choisie avant la synthèse du treillis).

Il est intéressant d'analyser le chemin de données conçu jusqu'à présent. Entre deux fronts montants d'horloge, une *Machine à état* envoie son RTS pour attendre un RTR, pour ensuite envoyer son instruction permettant aux opérandes de se rendre à l'ALU pour finalement acheminer le résultat vers le réseau de routeurs. Le décodeur est donc aussi utilisé pour rajouter au RTR la disponibilité de l'ALU et du réseau d'interconnexions. Cela signifie que si une des conditions n'est pas respectée, rien n'est acheminé à la $SALU$, et celle-ci attend un envoi (le flot de données se poursuit à travers le pipeline, mais il n'y a aucune nouvelle entrée).

### 3.5.2 ALU

Chacune des ALU permet de faire la consommation de deux jetons pour en former un nouveau, tout en mettant à jour les drapeaux d'état si désiré. Le travail se fait de manière purement combinatoire. Il est également important de mentionner que ce module ne voit que le décodeur ; tout le reste de la $SALU$ lui est transparent. Cette compartimentation simplifie le contrôle des jetons. Le jeu d'instructions se limite à 16 opérations, avec initialement 11 opérations combinatoires ; cela laisse 5 emplacements d'adressage pour qu'un utilisateur puisse instancier ses algorithmes. Par exemple, on pourrait vouloir faire le calcul de la racine carrée par une méthode itérative de Newton. Celle-ci peut être implémentée dans la SALU

Tableau 3.4 Affectation des drapeaux en fonction des opérations disponibles

| Instruction | Abréviation | Drapeaux affectés |
|---|---|---|
| Addition | ADD | Z-C-N-V |
| Soustraction | SUB | Z-C-N-V |
| Multiplication | MULT | Z-N-V |
| Décalage à gauche logique | SLL | Z |
| Décalage à droite logique | SRL | Z |
| Décalage à droite arithmétique | SRA | Z |
| Fonction logique ET | AND | Z |
| Fonction logique OU | OR | Z |
| Fonction logique OU exclusif | XOR | Z |
| Fonction logique NON | NOT | Z |
| Inversion des bits | FLIP | Z |

directement, ou de manière usuelle par l'entremise des mémoires de programme provenant des *Machines à états*. L'instauration d'une opération multi-cycles est permise en indiquant au décodeur la disponibilité de l'ALU. Le tableau 3.4 décrit les différentes opérations permises, ainsi que leur affectation sur les drapeaux d'états.

L'addition et la soustraction sont les seules opérations qui affectent les 4 drapeaux d'état. Bien que le Z et C soit assez simple, le drapeau de débordement signé est implémenté différemment selon la nature de l'opération. Pour une addition/soustraction, tout dépend du signe des opérandes et du résultat. Soit x le signe du premier opérande, y du deuxième et z du résultat, on retrouve pour le dépassement sur l'addition :

$$xy\bar{z} + \bar{x}\bar{y}z$$

Et pour le dépassement sur la soustration :

$$\bar{x}y\bar{z} + x\bar{y}z.$$

Le dépassement signé pour la multiplication est un peu différent car on va doubler le nombre de bits de chaque opérande. Il faut alors regarder les 32 bits excédentaires et le bit de signe (bit 31). Dans le cas où se sont tous des '1' ou des '0', il n'y a pas de dépassement. Le dernier drapeau d'état, celui de la valeur négative, est calculé de la façon suivante pour l'addition et la soustraction :

bit de débordement XOR bit de signe du résultat

Et pour le dépassement sur la soustration :

bit de signe de l'opérande 1 XOR bit de signe de l'opérande 2.

Les opérations disponibles sont implémentées de manière simple dans leur état initial, mais il est facilement envisageable d'utiliser toutes les techniques nécessaires afin d'obtenir de meilleures performances. En effet, un des objectifs principaux dans l'élaboration du treillis était d'avoir une fréquence maximale en fonction des opérations utilisées, de sorte à permettre l'accès aux ALU à l'utilisateur. Un décalage à barillet est utilisé pour plusieurs des opérations. Celui-ci permet, de façon purement combinatoire, de faire différents décalages. L'aspect "uni-cycle" est très important pour le treillis dans son état actuel car on veut obtenir un jeton résultat à chaque coup d'horloge. Néanmoins, l'architecture permet également les opérations multi-cycles.

### 3.5.3 Réseau de routeurs

Le réseau de routeurs est l'élément critique de la *SALU*. Il permet de faire transiger toutes les données vers leur destination en conservant leur ordre respectif. Le réseau doit pouvoir accepter 8 jetons par cycle (en provenance des 8 ALU existantes) et renvoyer le même nombre vers différentes destinations discutées précédemment. Le design proposé pour le réseau de routeurs se retrouve en figure 3.11. Il est constitué de 6 routeurs bien distincts, soit 4 selon le type 1 et 2 selon le type 2. En regardant la figure, on remarque que l'on privilégie les échanges horizontaux et les échanges verticaux. L'implémentation du flot de données traversant le treillis devrait majoritairement s'effectuer dans ces 2 sens. Lorsqu'on voudra faire passer une valeur autrement, il y aura un routeur de plus à traverser (donc un cycle supplémentaire). Néanmoins, le réseau reste pipeliné au cycle d'horloge. Ainsi, les routeurs de type 1 n'ont aucun contact entre eux et doivent obligatoirement passer par un routeur de type 2 pour s'échanger de l'information. Ces derniers doivent assurer cette bonne communication, tout en permettant les échanges horizontaux-verticaux. La figure démontre également que chaque routeur de type 1 est associé à un côté de la *SALU*. Il interagit avec les deux ALU ainsi que la *Banque de FIFO*. Les schémas haut-niveau des deux types de routeurs se retrouvent en figure 3.12.

Bien qu'il n'y est que deux types de routeurs (qui sont eux-mêmes semblables), chacun possède ses propres entrées et sorties. Le routeur 1_0 a donc comme sources les ALU 0 et 1 ainsi que le routeur 2_0. Les 3 FIFO représentées accueillent les jetons de 64 bits de ces trois instances. Les signaux de signalisation *full* des deux premières servent d'indication que la destination est disponible pour la *Machine à états* qui commandent les déplacements. De la même manière, le routeur 2_0 ne peut envoyer de jetons si la FIFO 3 est pleine. En sortie, on peut renvoyer les jetons vers les deux ALU dans le cas ou la donnée doit aller en tampon interne. Il faut cependant que les adresses soit pour celles attachées à ce routeur (soit ALU 0 et 1). Il en va de même pour un envoi vers une *Banque*. Si, par exemple, le jeton doit aller vers l'ALU 4, il faut faire passer le 64 bits vers le routeur 2_0 qui prendra ensuite la relève. Logiquement, il est impossible qu'un jeton arrivant dans la FIFO 3 retourne vers un routeur 2. La partie droite de la figure 3.12 démontre le même principe mais pour un routeur de type 2. On reçoit de l'information des routeurs adjacents horizontalement, et du deuxième routeur de type 2. Les sorties sont les mêmes, toujours en considérant qu'un jeton ne retourne pas au routeur qui l'a envoyé.

Les sous-modules "Unité de contrôle" représentent le coeur des routeurs. Lorsqu'une nouvelle valeur sort des FIFO, une vérification est faite sur la destination. Puisque chaque routeur

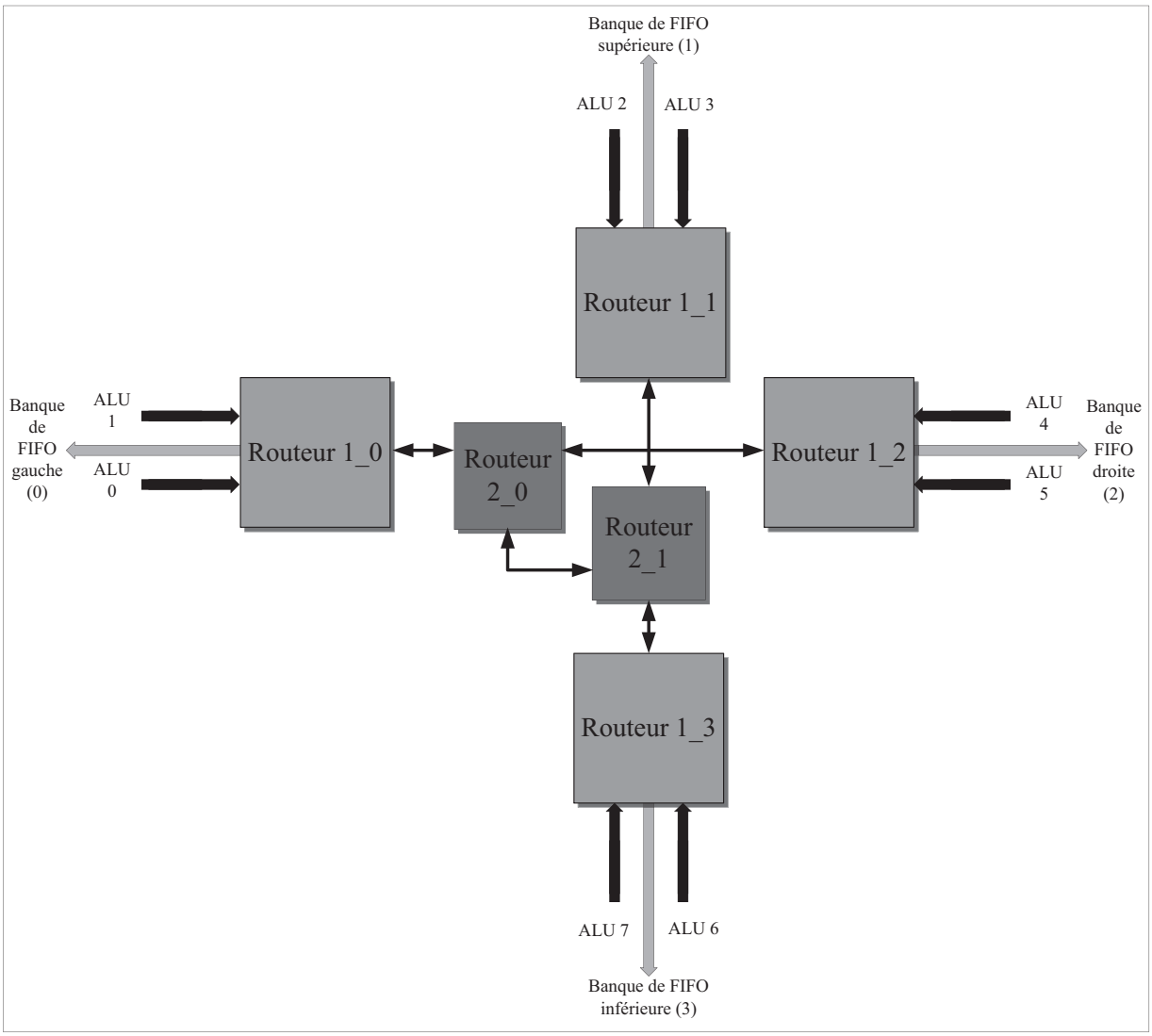


Figure 3.11 Réseau de routeurs

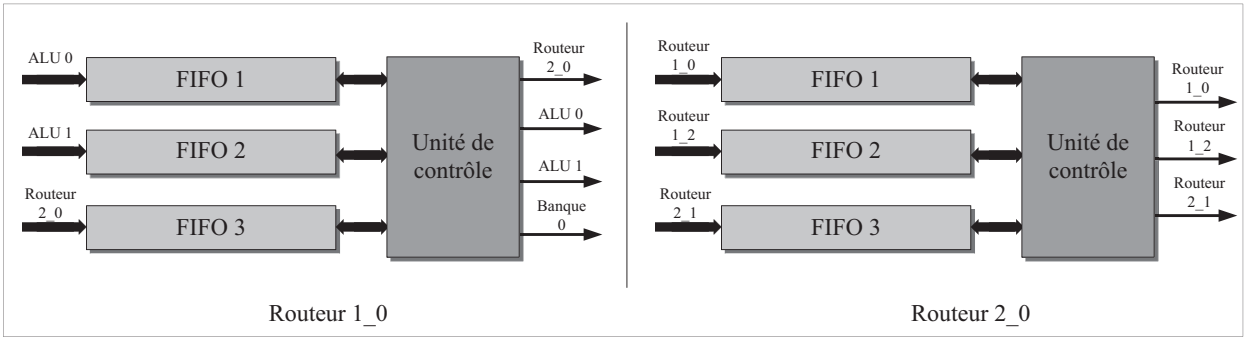


Figure 3.12 Schéma du routeur 1 et du routeur 2

a une identité propre, il est facile de savoir si le jeton en cours doit sortir du réseau ou bien y rester. Si la destination est atteinte, on élimine la partie de contrôle du jeton pour ne garder que la donnée sur 32 bits et l'inscrire dans son emplacement mémoire. Tous les signaux nécessaires provenant des FIFO en cause sont gérés par ce bloc.

Un problème survient cependant lorsque 2 ou 3 des FIFO de la figure 3.12 désirent envoyer en même temps un jeton vers le même endroit. En effet, bien qu'on puisse gérer les jetons de toutes les provenances vers toutes les destinations de façon concurrentielle, on doit conserver un débit d'un jeton par interface par cycle. Pour palier à ce problème, une première ébauche fut d'implémenter un algorithme de *round-robin* tout simple : un compteur cyclant sur les 3 FIFO. Ainsi, si plus d'une d'entre elles veulent accéder à la même sortie, le compteur décidera de celle qui aura l'accès. Bien que cette méthode intuitive puisse résoudre le problème, qu'arrivera-t'il si une seule FIFO demande l'interface alors que le compteur ne lui permet pas de faire le transfert ? On aurait des absences d'envoi sur certains cycles alors qu'il y a des jetons à transférer. Le compteur synchronisé sur l'horloge empêcherait de garder un bon débit à travers le réseau de routeurs. Le circuit de la figure 3.13 démontre la solution pour un exemple d'échange avec une *Banque de FIFO*. Si, par exemple, la FIFO 1 désire transiger un jeton, mais que le compteur ne lui permet pas, on examine les requêtes des autres FIFO afin de lui donner l'accès si possible. On s'assure de toujours avoir un débit constant (aucune machine à états ne vient ralentir les prises de décision) et de faire parvenir les données de sorte à boucler sur les différentes sources. En d'autres termes, les 3 FIFO sont analysées de sorte à les vider, dans le pire des cas, 1 fois par 3 cycles d'horloge.

En terminant, considérons le cas complet de la sortie d'une nouvelle instruction jusqu'à l'arrivée à destination du jeton final. Cette instruction demande une multiplication de deux jetons provenant de la *Banque de FIFO* vers une deuxième *Banque* située à la droite de la SALU désirée. La machine utilisée doit tout d'abord attendre de recevoir un RTR. Celui-ci est calculé en fonction de la présence des opérandes ainsi que de la disponibilité de la FIFO dans le routeur 1 joint à l'ALU. Par la suite, l'opération a lieu et le décodeur va sauvegarder ce jeton en mémoire dans le routeur 1. Puisque la destination se situe à la droite de la $SALU$, le jeton doit traverser les deux routeurs de type 2 ainsi qu'un nouveau routeur de type 1 pour finalement se diriger à bon port. Si le trafic ne cause pas de délai, le résultat arrivera 5 coups d'horloge après le départ de l'instruction. On voit ici que les FIFO présentes dans les différents blocs servent bel et bien d'étages de pipeline pour le chemin de données.

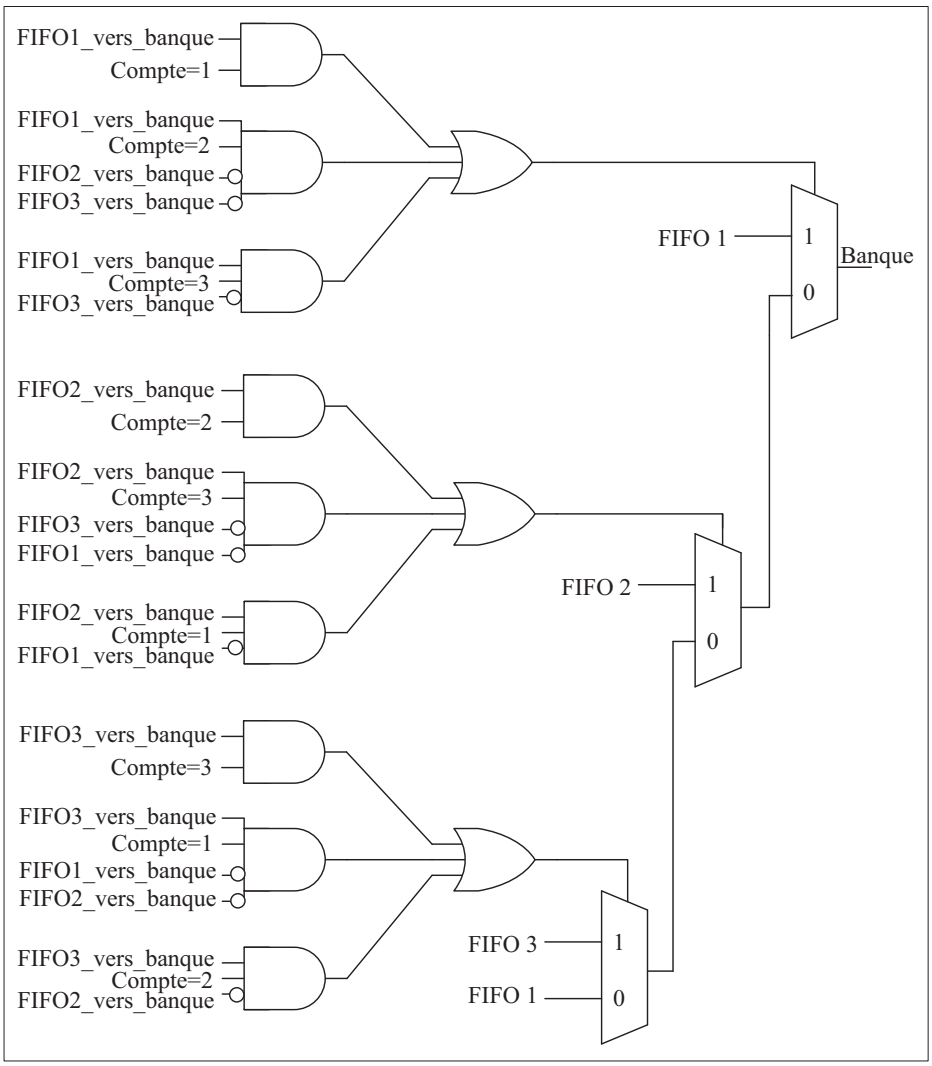


Figure 3.13 Circuit pour le *round-robin*

## 3.6 Conclusion

Dans ce chapitre, une solution basée sur le principe de treillis de calcul et de jetons de donnée est proposée afin d'adresser le besoin d'une architecture à deux niveaux. Au niveau matériel, le design permet non seulement de faire du traitement de signal à haut débit mais d'alléger la partie "contrôle" par l'entremise de jetons. Un programmeur purement logiciel peut venir, à haut niveau, instancier ses algorithmes complexes tout en ayant les bienfaits du parallélisme.

Les 3 modules distincts formant les tuiles du tissu sont finement construits de sorte à permettre la plus grande flexibilité possible pour une complexité moindre. Les *Machines à*

*états* contiennent les mémoires d'instructions du système. Elles représentent l'accès haut-niveau qui donne son sens au treillis. Elles communiquent avec des *SALU* à travers un réseau de *Banques de FIFO* qui emmagasinent les différents jetons, permettant ainsi de créer le flot de données. Les *SALU* proprement dites servent à faire tous les calculs nécessaires. Comme il a été prouvé à travers tous les modules, le pipeline et le chemin de données sont des éléments essentiels au bon fonctionnement des algorithmes implémentés.

Bien qu'un modèle *SystemC* fut construit préalablement pour rapidement déceler différents problèmes de conception et que le treillis résultant fonctionne correctement, celui-ci perd de sa valeur si l'implémentation d'algorithmes devient une tâche fastidieuse. En effet, le but ultime est de pouvoir écrire avec *MATLAB* (par exemple) les algorithmes et que la traduction vers le tissu se fasse automatiquement. Au stade actuel, le programmeur doit tout de même faire attention pour ne pas causer de problèmes internes. La courbe d'apprentissage est toutefois acceptable compte tenu des grands avantages et accélérations permis par la solution.

# CHAPITRE 4

# RÉSULTATS EXPÉRIMENTAUX

## 4.1 Introduction

L'implémentation de la solution proposée n'est pas une tâche aussi simple en soi. Bien qu'une analyse méticuleuse de l'état de l'art fut réalisée dans le but d'élaborer une architecture répondant aux besoins de l'industrie tout en évitant les pièges déjà répertoriés, la mise en oeuvre amène son lot de défis et de surprises. Un aspect important qui est souvent laissé pour compte aux premières étapes d'un projet est l'utilisation des ressources. Dans cette optique, plus grande est l'analyse du problème et l'ajout de fonctionnalités, plus il est important de percevoir le coût résultant. En effet, il suffit d'imaginer la situation où, au niveau de l'implémentation, on réalise qu'une fonctionnalité importante du circuit ne peut fonctionner sans l'ajout d'un bloc DSP. S'ils sont déjà tous utilisés, il faudra puiser dans les ressources disponibles sans quoi une réévaluation du design est incontournable. Il est même fréquent qu'une compagnie opte pour un FPGA plus puissant que celui choisi au départ pour rencontrer les requis. Le treillis fut donc fabriqué en ayant bien en tête les coûts de ressources encourus.

En premier lieu, nous présenterons l'implémentation du treillis sur FPGA. Cette section comporte tout d'abord une discussion sur la vérification du bon fonctionnement du design pour ensuite fournir les résultats et statistiques sur la plaquette de développement utilisée. Ensuite, nous verrons de quelle manière un algorithme de traitement de signal, plus précisément une DDC (*Digital Down Converter*), se transcrit sur la structure. Finalement, l'élaboration de la reconfiguration dynamique du treillis sera détaillée. Il est important ici de mentionner que les résultats expérimentaux seront plus qualitatifs que quantitatifs. La première phase fut tout simplement de faire fonctionner le circuit. Par la suite, l'amélioration de ses différentes facettes provient de plusieurs analyses en fonction des algorithmes implémentés.

## 4.2 Implémentation matérielle du treillis

Dès le départ du projet, l'idée était de décrire entièrement l'architecture de la solution en VHDL, ce qui fut réalisé, module par module. Rapidement, l'ampleur de la tâche se fit sentir et il fallu développer une méthode pour faire la vérification. On peut rapidement penser utiliser les sorties du modèle *SystemC* afin de faire une comparaison. Néanmoins, il faut avoir

une meilleure technique pour faire le déverminage si nécessaire. C'est dans cet état d'esprit que les stratégies de vérification furent élaborées. Commençons par définir les trois niveaux d'abstraction :

1. Niveau unitaire : permet de tester les différents blocs qui sont créés de façon individuelle. La taille de ces blocs est variable mais doit représenter un minimum de fonctionnalités pour en valoir la peine, sans être trop gros pour complexifier le test.
2. Niveau intégration : permet de vérifier le fonctionnement et la communication lors de l'assemblage des sous-modules.
3. Niveau système : permet de vérifier le comportement du circuit final.

Cette approche dite *bottom-up* est mise en pratique, dans la mesure du possible, par le flot de vérification représenté en figure 4.1. Le principal avantage du *bottom-up* est de pouvoir trouver les problèmes le plus vite possible pour les corriger plus facilement. Cela simplifie également l'intégration du système. La partie la plus importante du schéma est l'identification des cas de test. En effet, il faut bien connaître les fonctionnalités du design pour tester assez de cas et avoir confiance que le circuit fonctionne. Pour ce projet, les vérifications furent exécutées par l'entremise du logiciel ModelSim et de bancs d'essai sous le langage VHDL et *SystemVerilog*. Tout dépendant de la complexité des modules, un amalgame des techniques

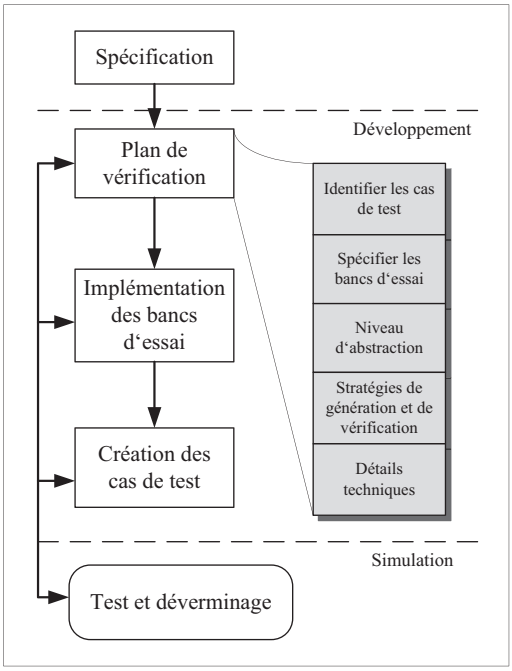


Figure 4.1 Flot de vérification

suivantes furent exploitées : couverture des états, des branches, du code, structurelle, fonctionnelle, assertions et tests aux limites. C'est de cette façon que le treillis fut vérifié et simulé, par une multitude de bancs d'essai aux trois niveaux d'abstraction. Ultimement, tous ces tests fournissent la conviction que l'architecture est fonctionnelle et opérationnelle avec un taux de satisfaction dépassant les 90%, ce qui est amplement pour les recherches que nous effectuons. Au niveau système, on peut donc voir que le treillis implémente bien les algorithmes. Au delà de ce point, une compagnie industrielle pourrait vouloir établir un plan de régression et même faire rouler des tests pendant plusieurs heures, voir jours. Il fut véridique, pour ce projet, que la relation 70%-30% entre le temps de vérification et de design fut observée. À titre d'exemple, voici quelques cas de test pour le routeur 1 présent dans la *SALU* :

- Vérifier que les adresses de destination soient correctement traitées pour les renvois ;
- Vérifier que la priorité en *round-robin* est respectée ;
- Vérifier le bon fonctionnement pour des grosseurs de mémoires différentes ;
- S'assurer que le routeur ne lit pas de donnée externe si la FIFO correspondante est pleine ;
- S'assurer que le protocole de communication avec le routeur 2 est respecté ;
- etc.

Une fois les simulations complétées et le bon fonctionnement observé, l'architecture fut implémentée sur FPGA. La plateforme utilisée pour ce faire est la carte DE3 de la compagnie Altera, possédant un Stratix III EP3SL150F1152C2N. Bien que cette puce ne soit pas parmi les plus récentes, c'est la plus performante disponible pour l'expérience et elle offre tout de même assez de ressources matérielles pour réaliser les tests. Pour former le treillis, on a besoin des trois modules (*Machine à états*, *Banque de FIFO* et *SALU*). Le coût des interconnexions va varier en fonction des grandeurs des rangées et des colonnes ; on peut donc se fier sur le résultat des 3 principaux blocs (voir tableau 4.1). Il est à noter que pour ces résultats, les *Banques* comportent 16 FIFO de 8*32bits. En premier lieu, on observe que les *Machine à états* consomment le moins de ressources, ce qui était attendu. Outre un peu de mémoire pour les instructions des programmes, ce module ne vient pas restreindre la grandeur maximale

Tableau 4.1 Ressources pour les 3 modules

| | Banque de FIFO | SALU | Machine à états |
|---|---|---|---|
| ALUT | 7444 | 11721 | 304 |
| ALM | 6538 | 8553 | 152 |
| Registres | 5920 | 7612 | 128 |

du treillis. Cette limite provient des deux autres blocs. En effet, on peut voir que le nombre d'ALUT, d'ALM et de registres est assez important. Pour la famille Stratix, le nombre de ressources se mesure principalement avec leur ALM (*Adaptive Logic Modules*). Chaque ALM contient une table de vérité à 8 entrées, 2 additionneurs dédiés et 2 registres. Le FPGA dont nous avons fait usage contient 57000 ALM. En reprenant les résultats du tableau 4.1, on en conclue qu'une *Banque de FIFO* utilise 13% des ALM disponibles, alors qu'une *SALU* en utilise 15%. Pris comme tels, ces chiffres sont de bonnes tailles et peuvent paraître comme un énorme inconvénient à utiliser cette architecture. Cependant, il ne faut pas oublier qu'on désire pouvoir faire de la reconfiguration dynamique, ce qui signifie que le synthétiseur ne peut simplifier les parties du circuit qui ne servent pas pour l'algorithme implémenté. En effet, en modifiant les instructions des programmes, on peut vouloir utiliser une ALU qui n'était pas en fonction jusque-là, d'où la nécessité de ne faire aucune simplification. Un usager qui ne désire pas avoir l'option de la reconfiguration pourra la désactiver et obtenir de faibles pourcentages d'utilisation. De plus, si on considère un FPGA plus récent, soit le Sratix IV GX qui contient 212480 ALM, les taux ci-haut s'abaissent à 3% et 4%. Pour un Stratix V GX possédant 359200 ALM, on obtient 1.8% et 2.4%. Ces chiffres pourraient même être plus bas en considérant que les ALM des dernières puces contiennent 6% plus de logiques disponibles qu'avant. On peut conclure que les ressources utilisées sont adéquates pour une architecture de ce type. Il est intéressant de voir que la limitation ne provient pas du tout de la mémoire ; nous n'avons même pas commencé à utiliser les blocs M9K et M144K présents dans le Stratix III. En utilisant ces blocs, on aurait encore plus de ressources disponibles pour le reste du circuit. Du côté de Xilinx, il est possible d'utiliser les tables de vérité inutilisées parmi les Slices pour en faire des mémoires et ainsi avoir un design encore plus optimisé au niveau des coûts. On en vient à la conclusion que plus le FPGA est gros, plus notre architecture sera profitable.

## 4.3 Implémentation d'algorithmes

Pour proprement juger notre architecture NoC, il faut pouvoir y implanter des algorithmes. La majorité des opérations que le treillis doit supporter dans le domaine du traitement de signal est l'arithmétique répétitive. On entend par cela que le débit de données à l'entrée est quasi-constant et qu'on doit pouvoir faire le même jeu d'opérations pour chaque nouveau jeu d'entrées. L'exemple le plus simple est celui de la moyenne, que nous avons représenté précédemment sur la figure 3.3. Les mêmes instructions se font cycle après cycle tant que le pipeline est plein et tant que des opérandes sont fournis à l'entrée.

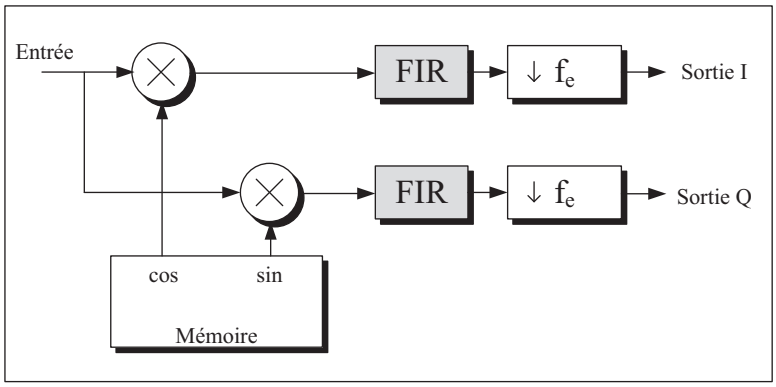


Figure 4.2 Schéma d'une DDC

En premier lieu, plusieurs petits algorithmes simples furent utilisés sur le treillis. Le premier défi provient de l'arrivée des jetons. En effet, sur un produit final, le treillis viendra se brancher à des mémoires périphériques qui recueilleront et fourniront des données. Pour nos besoins expérimentaux, celles-ci furent initialement pré-encodées dans des FIFO présentes en début de parcours. Par la suite, un générateur matériel pseudo-aléatoire de données vint fournir le débit recherché en entrée. En sortie, nous pouvons aller lire la dernière FIFO de résultats pour faire les comparaisons avec le calcul théorique (qui est simple à calculer connaissant les entrées). L'utilisation de l'outil SignalTap d'Altera, qui permet de capturer des informations sur la puce pour les renvoyer par JTAG vers l'ordinateur, aida pour valider les résultats. Une autre méthode, qui fut grandement utilisée, est de venir brancher le processeur NIOS II d'Altera pour avoir à notre disposition une console pour afficher ce que l'on désire. Nous étayerons cette technique dans la prochaine section. L'algorithme sur lequel nous nous pencherons est la DDC, dont le schéma se retrouve sur la figure 4.2. Les valeurs de cosinus et sinus (synthétiseur) peuvent facilement être mises dans une table de vérité, et la baisse de fréquence d'échantillonnage s'effectue en laissant tomber certains échantillons. Le filtre FIR, cependant, est un beau défi à implémenter et est utilisé à travers une multitude d'algorithmes comme celui-ci. Par soucis de simplification et de clarté, l'implémentation sur treillis que nous analyserons est celle d'un filtre FIR, dont la formule est :

$$y_n = \sum_{i=0}^{N} C_i \times x_{n-i},$$

où $y_n$ est le signal de sortie, $x_n$ est le signal d'entrée, $C_i$ sont les coefficients des branches (*tap*) et N est l'ordre du filtre (nous utiliserons N=2 pour la démonstration). Sur la partie supérieure de la figure 4.3, on retrouve une représentation graphique classique d'un FIR pour 3 jetons d'entrée. Les FIFO situées entre les opérateurs permettent d'accumuler les différents

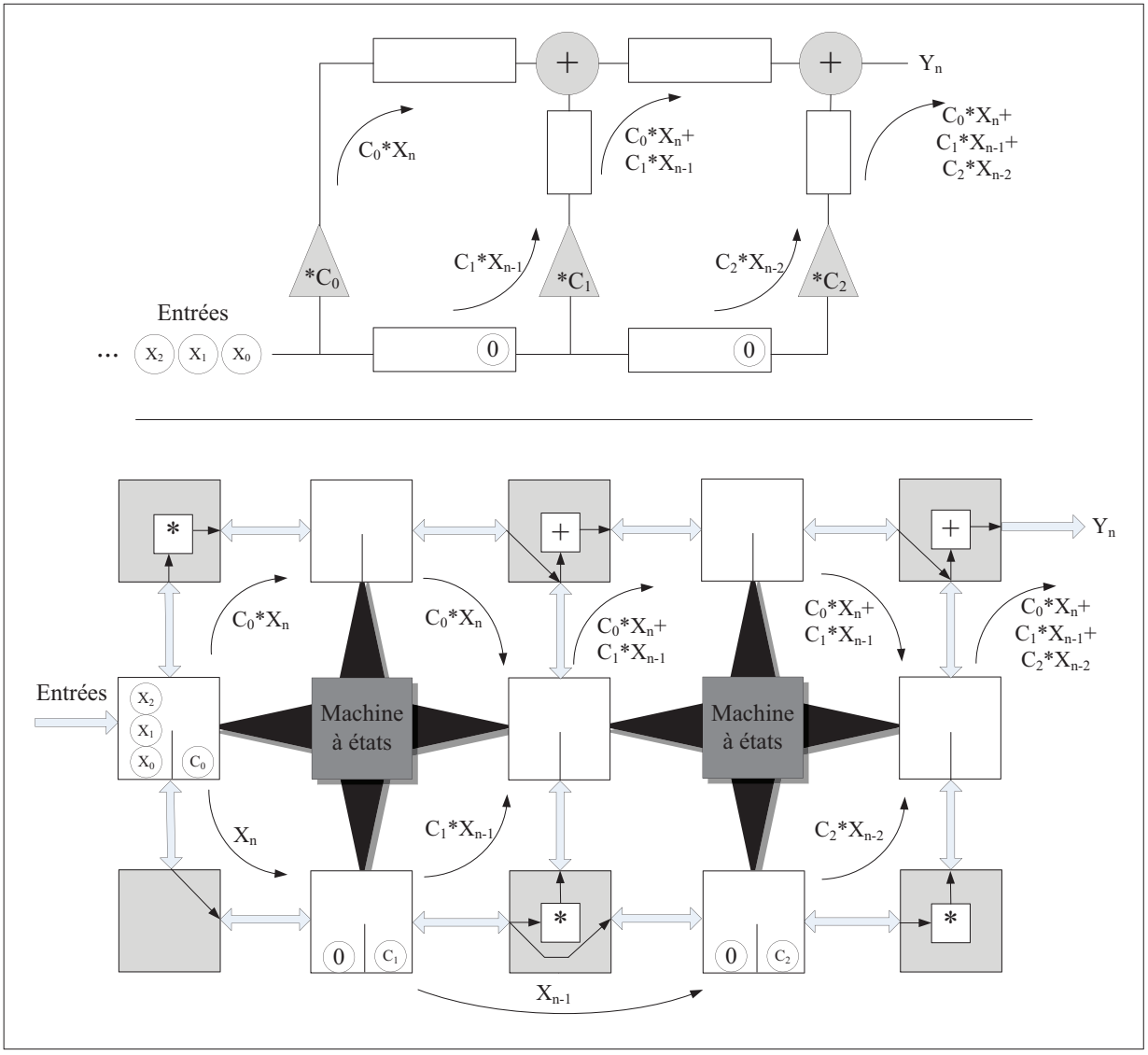


Figure 4.3 Filtre FIR implémenté sur le treillis

jetons de résultat. Pour créer le délai nécessaire entre chaque branche, des jetons nuls sont initialement insérés dans les FIFO. Ceci signifie qu'il n'y a aucune synchronisation à faire et ce, peu importe le nombre de branches que l'on désire ajouter. Il suffit tout simplement de mettre un jeton nul par FIFO inférieure. Une implémentation en VHDL pur aurait au moins nécessitée un registre à décalage pour faire la même chose. En se référant à l'image, on aurait une pile d'un seul jeton nul dans la première fifo verticale et deux pour la deuxième après 2 cycles. Le pipeline se remplit au complet et en régime permanent, nous avons une nouvelle entrée et une nouvelle sortie à chaque cycle d'horloge.

L'implémentation sur le treillis se retrouve sur la partie inférieure de la figure 4.3. On peut facilement voir la corrélation avec la représentation classique. Une différence notoire est l'ajout des jetons de coefficients. Ceux-ci sont pré-programmés (tout comme les jetons nuls) et ne sortent jamais des FIFO ; on utilise donc une instruction de lecture profonde. Les autres jetons circulent selon les instructions indiquées. La *Machine à états* de gauche, par exemple, utilise cinq de ses huit programmes disponibles :

- Programme 0 : Transporter $C_0{}^*X_n$ ;
- Programme 4 : Transporter $X_{n-1}$ ;
- Programme 5 : Calculer $C_1{}^*X_{n-1}$ ;
- Programme 6 : Transporter $X_n$ ;
- Programme 7 : Calculer $C_0{}^*X_n$.

Les machines 5 et 7 font chacune une multiplication pour acheminer le résultat vers une nouvelle *Banque de FIFO*, alors que les autres ne font que transporter des jetons. Il est important de comprendre que le programme reçoit uniquement l'emplacement des opérandes ; il n'a aucune conscience de leur valeur comme telle. Ainsi, il est très simple de prendre ces mêmes instructions et de les retranscrire pour une infinité théorique de branches. Un des buts importants du projet est effectivement de pouvoir facilement agrandir le treillis afin d'augmenter soit le débit d'entrée, soit la complexité des algorithmes. De plus, chacun des programmes ne contient qu'une seule instruction qui est appliquée indéfiniment. Cette situation va s'appliquer régulièrement car, comme nous l'avons mentionné précédemment, les algorithmes de traitement de signaux sont souvent répétitifs. C'est un avantage pour notre design. Mentionnons également que le pipeline se remplit peu à peu et que les instructions attendent patiemment leur jetons d'opérande. Donc, les instructions provenant de la *Machine à états* de droite n'ont initialement pas lieu par manque de données, ce à quoi nous nous attendons.

Les résultats que l'on tire de cette implémentation ont un sens important au niveau des objectifs mis en place en début de projet. L'implémentation logicielle de l'algorithme sur le treillis fixe, qui se fait très simplement, donne comme résultat le bon fonctionnement de notre exemple d'un filtre FIR (ou peu importe l'algorithme en question). Sans trop connaître les détails de l'architecture, il est possible de profiter de la puissance des FPGA et d'une transcription matérielle sans difficulté. Qui plus est, l'absence de reconfiguration dynamique permet d'avoir un très faible taux d'utilisation de ressources, tout en ayant eu qu'à programmer les mémoires. Les tests sur plaquette avec un filtre d'ordre 4 fournissent bel et bien un nouveau résultat par cycle d'horloge. Pour ce faire, il faut utiliser les multiplieurs 18 bits

présents sur la puce qui donne des résultats aux cycles. Cette grandeur d'opérandes de 18 bits n'est pas aléatoire et provient des requis du partenaire industriel. Il n'y a donc aucune perte dans le pipeline instauré, et ce pipeline sera le même pour tous les algorithmes puisque le treillis est fixe. La fréquence maximale fournie par le logiciel Quartus II est de 204.75MHz. Celle-ci va dépendre en grande partie des opérateurs présents dans les $SALU$, et très peu des algorithmes implantés. Il est donc très simple de la contrôler, peu importe la situation.

Un autre avantage majeur qu'il faut souligner est qu'il est possible, avec le treillis de la figure 4.3, d'ajouter un deuxième algorithme, totalement indépendant du premier, sans perturber le flot. Par exemple, le deuxième filtre FIR de la figure 4.2 peut débuter à gauche pour aller vers la droite, ou bien celui déjà présent se plier sur lui-même. Le treillis est effectivement construit pour favoriser les directions gauche-droite, haut-bas. La planification des nouvelles FIFO (provenant des *Banques*) et des instructions-programme demeure toutefois un défi en soi. En effet, il faut s'assurer de ne pas créer de goulots d'étranglement à travers un canal particulier. En prenant en considération que les opérateurs non utilisés jusqu'à présent ne sont pas simplifiés lors de la synthèse, il serait favorable de les utiliser. Ces derniers exemples d'obstacles ne sont cependant rencontrés que si l'on désire conserver le même débit. Dans le cas contraire, on peut ralentir la cadence et faire nos opérations dans l'ordre que l'on veut en gérant correctement les différentes FIFO. Un futur outil de synthèse logiciel permettrait de prendre les meilleures décisions de placement, l'utilisateur n'ayant même plus à prendre ces considérations.

## 4.4 Reconfiguration dynamique

Généralement, la redéfinition d'un circuit logique au sein d'un FPGA est une opération singulière, peu fréquente et réalisée alors que le système est "hors-fonction". Suivant les progrès technologiques, une nouvelle tendance au sein du domaine reconfigurable est de pouvoir modifier ou même changer totalement le comportement du circuit sans en interrompre le fonctionnement. On discerne rapidement les avantages d'une telle possibilité. En effet, le concepteur peut facilement adapter son application faisant face à de nouvelles contraintes pour fournir un algorithme plus efficace. Par exemple, on peut imaginer comme contrainte une fluctuation du débit d'entrée. Il serait intéressant de pouvoir rediriger le trafic de façon dynamique selon les besoins, ou même de modifier le comportement des sauvegardes en mémoire. Le matériel devient extrêmement malléable et contrôlable par logiciel et peut être utilisé à des instants différents pour exécuter des opérations différentes. On dispose donc quasiment de la souplesse d'un système informatique qui peut exploiter successivement des

programmes différents, mais avec la notion fondamentale d'une configuration matérielle nettement plus puissante.

Notre architecture se veut fixe, c'est-à-dire qu'une fois la taille choisie, on ne devrait plus vouloir modifier le placement et le comportement des modules présentés antérieurement. Néanmoins, deux possibilités intéressantes de reconfiguration s'offrent à nous : celle du contenu des FIFO et celle des mémoires d'instruction. Dans le premier cas, on parle de venir altérer la valeur des constantes dans les *Banques de FIFO*. En reprenant notre exemple du filtre FIR, on pourrait réviser les coefficients des différentes branches. Le deuxième cas sur lequel portera notre expérimentation est la modification du contenu des mémoires de programme présentes dans les *Machines à états*. L'outil SOPC Builder sera utilisé afin de fabriquer le système nous permettant de faire de la reconfiguration. Un schéma de ce système se retrouve en figure 4.4. La pièce centrale est le processeur NIOS II *cpu_0* et va interagir avec les différents modules, le tout excité par l'horloge *clk_0* de 50MHz. À travers le bus Avalon, le bloc *jtag_uart_0* permet de faire des communications par JTAG. Les modules *timer_0* et *sysid* servent au bon fonctionnement du processeur. La mémoire utilisée dans cet exemple (*onchip_memory2_0*) est directement sur la puce et joue le rôle de mémoire de don-

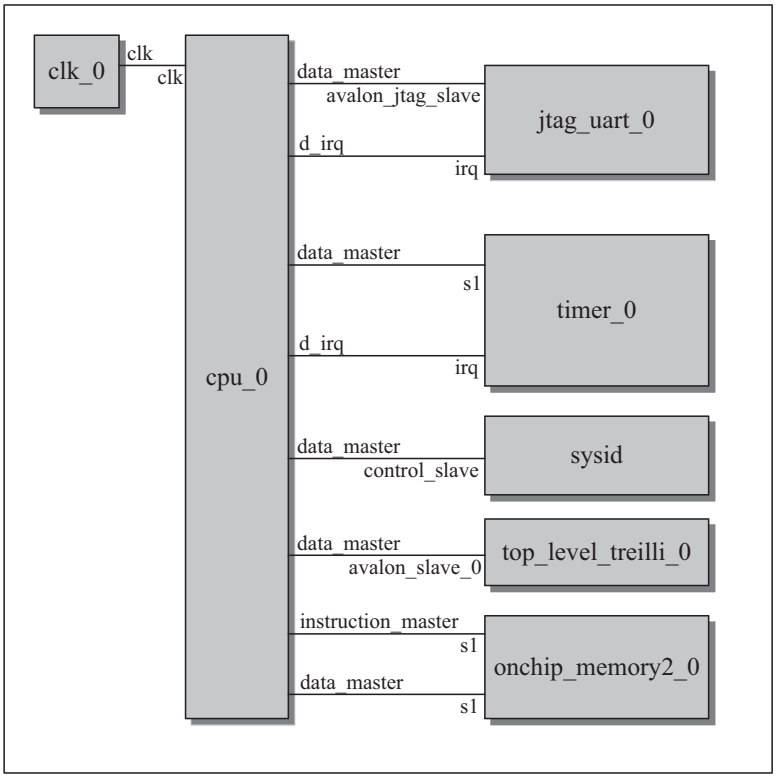


Figure 4.4 Système généré par SOPC

Tableau 4.2 Interface vers le treillis

| Nom | Taille | Direction |
|---|---|---|
| addr_write | 7 bits | entrée |
| data_to_write | 32 bits | entrée |
| data | 32 bits | sortie |
| wr | 1 bit | entrée |
| rd | 1 bit | entrée |

Tableau 4.3 Description de l'adressage

| Bits | Description |
|---|---|
| 6 | Active ou désactive la transmission des instructions |
| 5-3 | Numéro de la machine (programme) |
| 2-1 | Emplacement de l'instruction |
| 0 | Partie haute ou basse de l'instruction |

nées et d'instructions de programme (pour *cpu_0*). En dernier lieu, nous avons notre treillis (*top_level_treilli_0*) qui est branché en temps qu'esclave sur le bus. De ce fait, Altera fournit une interface qui nous permettra d'aller lire et écrire sur ce coprocesseur. Les écritures serviront à remanier les valeurs des mémoires d'instructions, tandis que les lectures fourniront les résultats des calculs. À cette fin, Les signaux utilisés de l'interface sont cités dans le tableau 4.2.

Pour faire une écriture, on doit envoyer un signal d'activation (*wr*), des données à envoyer (*data_to_write*) et une adresse (*addr_write*). Pour la preuve de concept, le treillis utilisé ne comporte qu'une seule *Machine à états* avec de petites mémoires, mais les mêmes techniques peuvent s'appliquer pour une architecture plus volumineuse en modifiant tout simplement le format de l'adressage du tableau 4.3. Le bit 6 est utilisé pour indiquer qu'il y a un transfert de données provenant du bus et qu'on doit arrêter le travail en cours. Les nouvelles instructions viennent remplacer celles présentes dans la machine (bit 5 à 3) et à l'emplacement (bit 2 à 1) voulu. Le bit 0, quant à lui, permet de choisir entre la partie haute et la partie basse de l'instruction. En effet, nous utilisons des mots de 47 bits, mais le processeur NIOS II ne peut transférer que 32 bits à la fois. Finalement, le signal *rd* permet de renvoyer, au cycle suivant, la sortie de la FIFO sur laquelle il est branché par *data*. Cette FIFO est ajoutée au treillis pour ne pas venir déranger les interfaces déjà présentes.

Afin d'opérer le bon fonctionnement de ces signaux, nous utilisons un programme logiciel qui se situe au niveau du processeur *cpu_0*. Celui-ci va lire quelques résultats, modi-

fier le comportement de l'algorithme implémenté et lire de nouveaux résultats. Pour parvenir au treillis par coups de 32 bits, nous utiliserons les instructions IOWR_32DIRECT et IORD_32DIRECT. L'exemple suivant démontre une séquence d'envois au treillis :

```
1  IOWR_32DIRECT(TOP_LEVEL_TREILLI_0_BASE,4*64,0);
2
3  IOWR_32DIRECT(TOP_LEVEL_TREILLI_0_BASE,4*((6<<3)+(0<<1)+0),1108355232);
4  IOWR_32DIRECT(TOP_LEVEL_TREILLI_0_BASE,4*((6<<3)+(0<<1)+1),2064);
5  IOWR_32DIRECT(TOP_LEVEL_TREILLI_0_BASE,4*((6<<3)+(1<<1)+0),1108355232);
6  IOWR_32DIRECT(TOP_LEVEL_TREILLI_0_BASE,4*((6<<3)+(1<<1)+1),2064);
7  IOWR_32DIRECT(TOP_LEVEL_TREILLI_0_BASE,4*((6<<3)+(2<<1)+0),1108355232);
8  IOWR_32DIRECT(TOP_LEVEL_TREILLI_0_BASE,4*((6<<3)+(2<<1)+1),2064);
9  IOWR_32DIRECT(TOP_LEVEL_TREILLI_0_BASE,4*((6<<3)+(3<<1)+0),1108355232);
10 IOWR_32DIRECT(TOP_LEVEL_TREILLI_0_BASE,4*((6<<3)+(3<<1)+1),2064);
11
12 IOWR_32DIRECT(TOP_LEVEL_TREILLI_0_BASE,4*64,0);
```

À la ligne 1, on met seulement le bit 7 à '1' pour indiquer un début de transfert. Puisque les données se transmettent par octet, nous devons toujours faire une multiplication par 4 afin d'adresser le bus correctement (sur 32 bits). Les lignes 3 et 4 permettent de venir modifier un premier mot d'instruction de la troisième machine à l'emplacement 0. Lorsque nous mettons le bit le moins significatif de l'adresse à 0, nous envoyons les 32 bits les moins significatifs de la nouvelle instruction. Dans le cas contraire, on envoie la séquence restante (15 bits). Ainsi, il faut faire deux envois successifs pour faire une modification. Suivant la fin des changements voulus, on vient indiquer au treillis qu'il peut entamer ses nouvelles tâches (ligne 12).

La conclusion de cette expérience est positive. Le test effectué se base sur la partie de l'algorithme de la figure 3.3 où l'on effectue de simples additions. En connaissant le contenu des mémoires de programme, la technique décrite précédemment permet de venir modifier le comportement de l'algorithme. La partie logicielle située au niveau du processeur récolte quelques valeurs de somme pour ensuite dicter un changement vers des soustractions. Lors de la nouvelle prise de données, nous obtenons bel et bien les différences attendues. Néanmoins, certaines considérations sont de mises pour assurer un déroulement adéquat. Une étape essentielle lorsqu'on indique une reconfiguration est d'attendre que le pipeline se vide aux étages situés entre les *Banques de FIFO*, soit aux *SALU*. Dans le cas contraire, les résultats d'opérations qui se dirigent vers les *Banques* proviendraient toujours des additions et il serait impossible de savoir à quel niveau on commence à recevoir les retours de soustraction. Dans

le même ordre d'idée, il est préférable, si possible, de changer toutes les FIFO (des *Banques*) pour de nouvelles dans le but de s'assurer de ne pas créer de conflits. L'instruction de *flush* est également de mise. Ce type de problème est bien connu dans le domaine de la reconfiguration dynamique, où la transmission des résultats intermédiaires d'une configuration à l'autre est difficile à coordonner. Cette situation a un impact important sur le processus de conception car toutes les configurations doivent être soigneusement conçues, de façon à ce qu'elles ne puissent fonctionner qu'à travers les différentes phases de l'application, et qu'elles puissent communiquer les résultats intermédiaires entre elles.

## 4.5 Comparaison avec des architectures connues

Afin de véritablement tester la valeur d'une architecture, il est toujours intéressant de pouvoir la comparer avec des designs qui font la même chose. Dans le cas de notre SDFPGA, cette tâche n'est pas si simple que ça. Le champ d'expertise et les requis d'application sont si pointus qu'il est quasi-impossible de trouver un treillis de calcul qui peut réaliser les mêmes choses que le nôtre. Nous avons déjà établi les nombreux avantages liés à l'utilisation de notre architecture ; nous tenterons ici de faire des analogies avec les NoC et leurs forces et faiblesses présentées en revue de littérature. Il y a de fortes ressemblances avec le RaPiD au niveau de la structure statique sur laquelle un contrôle dynamique vient faire le choix des opérations et des opérandes. De plus, les auteurs de l'architecture visent des applications à très haut débit. Cependant, leur treillis gère mal les tâches structurées et répétitives et se transpose mal dans un environnement 2D. Notre architecture ne souffre pas de ces contretemps et offre de bonnes performances au niveau du parallélisme. Du côté de PipeRench, on parle d'un ordonnancement en temps réel, ce que permet également notre solution. Ce design a toutefois les mêmes problèmes que le RaPiD, en plus d'avoir un routage très couteux pour relier tous les blocs aux I/O. Dans notre cas, nous avons porté une attention particulière à ce détail en ne donnant accès aux entrées et sorties qu'à certaines tuiles localisées au pourtour du treillis. Il est beaucoup plus stratégique de limiter ces accès pour simplifier le travail. Pour l'architecture RAW, les inconvénients relatés dans ce mémoire (problèmes lorsqu'il y a beaucoup de synchronisation entre les données, grande consommation de ressources, aucune possibilité d'instructions spécialisées) ne se retrouvent pas dans notre réalisation. Bien entendu, l'utilisation des ressources est tout de même importante dans notre cas, mais nous avons démontré que pour de grands FPGA, on pourrait mettre plusieurs tuiles et former un bon treillis. Pour MorphoSys, la différence majeure est qu'il faut remodeler l'architecture en fonction des applications, ce qui n'est pas pratique. De plus, leur design n'est pas conçu pour les FPGA. Les nouvelles tendances (comme le EGRA) poussent l'utilisateur à faire du

placement pour trouver une topologie bénéfique selon leurs besoins. Il faut alors être un expert dans le domaine, ce qui ne s'inscrit pas dans nos requis. En ayant une topologie fixe, nous nous assurons de camoufler la partie matérielle au développeur potentiellement uniquement logiciel. Finalement, notre reconfiguration dynamique permet de modifier le comportement du circuit, mais nous rencontrons le même type de problèmes répertoriés en littérature.

## 4.6 Conclusion

Dans ce chapitre, les résultats expérimentaux découlant de nos recherches ont été présentés. Une vérification exhaustive fut initialement réalisée afin de capter dès que possible les problèmes provenant de l'architecture. Il est ainsi possible d'apporter des modifications sur le design. Par la suite, l'implémentation matérielle offre de bons résultats sur les FPGA de dernières générations. Ceux-ci offriront encore plus de ressources qui permettront à notre treillis de prendre toute sa force et sa performance. Le filtre FIR est un bon exemple pour faire ressortir les capacités de l'architecture, et son implémentation démontre non seulement le bon fonctionnement de notre NoC mais également qu'un débit constant et qu'un pipeline bien rempli sont possibles. Les différents points dégagés à ce sujet attestent de la valeur du treillis de calcul. Finalement, la reconfiguration dynamique permet de modifier l'allure initiale du flot de données par une modification des instructions. Bien entendu, les accomplissements ne s'inscrivent pas parfaitement dans la définition même d'une reconfiguration car la logique reste la même. Cependant, on vient tout de même remanier le comportement du design de façon dynamique par l'entremise d'une séquence logicielle. Le treillis conçu offre de belles opportunités pour combler les manques dans le monde des architectures reconfigurables.

# CHAPITRE 5

# CONCLUSION

## 5.1 Synthèse des travaux

Dans ce travail, un treillis de calcul reconfigurable permettant de faire du traitement numérique a été élaboré. Cette architecture est composée de trois blocs distincts positionnés de façon stratégique de sorte que le chemin de contrôle et le chemin de données soient les plus efficaces possibles. En premier lieu, le module *Machine à états* représente l'accès entre l'utilisateur et le treillis. C'est à ce niveau que l'on indique la provenance des données utilisées et les chemins à prendre à travers l'architecture. Ces données sont sauvegardées sous forme de jetons. Ils ont l'avantage de grandement simplifier la synchronisation entre les entités du design et de diminuer les ressources utilisées par le chemin de contrôle. Ces jetons parcourent notre NoC à travers les *Banques de FIFOs* qui représentent les fils de transport du trafic. Il suffit de les adresser en fonction de la direction voulue. Le dernier module est la *SALU* et permet de faire tous les calculs nécessaires. Des opérations de base peuvent être effectuées (addition, soustraction, etc.), mais également des algorithmes plus complexes sous forme d'instructions spécialisées (racine carrée, etc.). La beauté de cette approche est que peu importe le temps que peut prendre une opération, le treillis restera synchronisé car on attendra les jetons de résultats avant de poursuivre. Nous avons donc la garantie de toujours avoir les sorties le plus rapidement possible. Le pipeline optimisé est la preuve de cette affirmation. On peut voir notre réalisation comme une grande étendue d'opérations déclenchées par l'arrivée de jetons qui s'y rattachent. La création et la consommation de ces jetons permettent ultimement de fournir un résultat final par cycle d'horloge.

La première étape d'expérimentation fut de fournir le pourcentage d'utilisation des ressources. Tel qu'attendu, les mémoires sont peu sollicitées et ne viendront pas limiter la grandeur du treillis. Celui-ci devient bénéfique pour de grands FPGA, ce qui était le but dès le départ du projet. En effet, un des objectifs était d'utiliser à bon escient la logique programmable qui se fait de plus en plus abondante. Nous avons ensuite détaillé l'implémentation d'un filtre FIR utilisé dans une DDC. La simplicité d'utilisation des différentes mémoires d'instructions permet de donner accès au parallélisme de la technologie FPGA à un niveau logiciel. Les résultats montrent le bon fonctionnement de l'algorithme avec un débit d'une sortie par cycle d'horloge (pour un pipeline plein). De plus, le treillis peut encore accepter des

applications supplémentaires, tant que la gérance du trafic se fait correctement. La fréquence maximale approchant les 200MHz ne dépend pas des algorithmes implantés mais plutôt des opérations utilisées, ce qui permet de mieux cibler les éventuelles optimisations permettant de rencontrer les requis. La dernière étape expérimentale consistait en l'élaboration d'une technique de reconfiguration dynamique de l'architecture. En utilisant un processeur NIOS II pour faire les tests, il fut possible de venir commander au treillis de modifier son comportement. En sommaire de ces différents résultats, nous pouvons affirmer qu'une nouvelle méthodologie permettant de contrôler l'énorme puissance de calcul d'un FPGA fut créée. Le treillis permet de masquer les détails et la complexité de l'implémentation matérielle tout en conservant une flexibilité accessible au niveau logiciel. Bien entendu, le but n'était pas d'utiliser les ressources sans jugement et que selon leur disponibilité, mais également d'avoir une architecture optimisée et performante, ayant un caractère générique et démontrant une facilité de mise en oeuvre. Ces objectifs furent comblés à travers notre SDFPGA.

Plusieurs des architectures étudiées en littérature offrent de bonnes performances et apportent de nouvelles idées pour tenter de faciliter l'exploitation du parallélisme, mais plusieurs petits bémols viennent poser des limites à leur utilisation. Notre propagation d'informations de contrôle est simple et efficace, tandis que notre chemin de données est assez flexible pour réaliser une majorité des requêtes possibles. De plus, il est encore possible de venir brancher un processeur à usage général ou de la mémoire supplémentaire sur le treillis pour donner plus de fonctionnalités sans vraiment altérer le mode de fonctionnement du système. La vérification au niveau matériel étant déjà complétée, les causes de problèmes seront reliées au niveau logiciel et seront facilement identifiables. Cependant, il faut être bien prudent avec ce genre d'architecture afin d'éviter que son utilisation ne devienne compréhensible que pour des experts. Une société américaine située en Oregon, dont l'unique produit était un FPOA (*Field Programmable Object Array*), fit rapidement faillite. Leur invention promettait un bel avenir, mais son utilisation beaucoup trop complexe vint tuer le projet. Comme il fut discuté précédemment, les outils de compilation haut-niveau sont actuellement très peu flexibles et ne permettent pas de prendre n'importe quel algorithme pour simplement le transposer au niveau matériel. C'est une variante de cette caractéristique qui donna un dur coup pour la popularité de ce type d'architectures et il est important de s'assurer qu'un support logiciel adéquat sera disponible pour l'utilisation du produit.

## 5.2 Limitations de la solution proposée

Bien entendu, notre solution proposée n'échappe pas à la règle et laisse transparaître quelques petites failles et limitations. Ce n'est rien qui peut empêcher le bon fonctionnement

du SDFPGA, mais il est toutefois important de les connaître avant d'utiliser l'architecture. En premier lieu, la facette du treillis qui demande une très bonne compréhension est celle du choix des FIFO utilisées (à travers les *Banques*). Les résultats viendront s'y empiler ; il faut donc être prudent de ne pas trop charger les mêmes FIFO car on pourrait avoir des résultats d'opérations différentes s'entremêler, sans que l'on ne sache d'où ils proviennent. Une façon de contrer ce problème serait d'utiliser des FIFO colorées qui pourraient pairer les différentes opérandes pour éviter les mauvais amalgames. Par la suite, il n'y a présentement pas de filet pour capter les programmes qui pourraient causer des *deadlock*. Si une instruction attend qu'une FIFO se remplisse alors qu'il n'y a aucun programme qui la contrôle, on attendra indéfiniment un jeton et le flot de données sera bloqué. C'est donc au programmeur de bien indiquer aux différentes petites machines comment faire circuler les jetons. Ensuite, au niveau matériel proprement dit, nous avons indiqué un retour maximal possible de 2 jetons par cycle d'horloge vers chacune des *Banques*. Il peut donc arriver qu'un ralentissement ait lieu si on tente de faire passer trop de trafic par les mêmes tuiles. Il est possible de modifier ce nombre d'échanges pour accommoder un algorithme, mais il est mieux de restructurer le passage des jetons. Finalement, au niveau de la reconfiguration dynamique, nous avons discuté du problème de vider le pipeline entre les modifications pour s'assurer de ne pas conserver les jetons qui n'ont plus de signification. Il faut prendre quelques coups d'horloge supplémentaires pour que les *SALU* terminent leur travail car nous n'y avons pas accès par le processeur. Cette embûche est bien connue dans le domaine. Il faut faire attention de ne pas amorcer une reconfiguration lorsque le débit d'entrée est grand car on pourrait perdre plusieurs données.

## 5.3 Améliorations futures

Pour conclure cet ouvrage, nous discuterons des différentes possibilités d'avenir pour le SDFPGA qui fut élaboré et conçu. Un aspect important que nous avons peu mentionné est la nécessité d'un compilateur qui permettrait de passer d'un langage connu vers notre treillis. Pour l'instant, il faut utiliser un *package* VHDL pour venir écrire à la main chacune des instructions sous le format des 47 bits. En offrant une méthode de compilation pour passer d'un langage tel que MATLAB ou C vers ce que le treillis supporte, il serait possible de régler la plupart des limitations discutées précédemment. Qui plus est, l'objectif ultime serait que l'utilisateur n'ait même plus à donner d'indications sur les différents choix à faire sur le treillis et laisser tout ce travail au compilateur. On aurait ainsi réellement une implémentation matérielle transparente à l'usager.

Une autre avenue fortement intéressante est celle des réseaux ou grappes de FPGA. Plusieurs recherches existent sur ce sujet et tentent de fournir une communication inter-puce assez

puissante pour permettre d'avoir accès à une quantité de ressources bien plus grandes. Plusieurs prototypes commerciaux sont fort intéressants, comme la BEE3 qui est une plaquette composée de 4 Virtex-5, 64GB de mémoire DDR2 et 8 ports 10GigE pour des communications internes. Sur ce type d'architecture, notre treillis pourrait atteindre des performances très impressionnantes et permettre de gérer plusieurs algorithmes en offrant un taux de parallélisme remarquable.

# RÉFÉRENCES

ALLARD, M., GROGAN, P., SAVARIA, Y. et DAVID, J.-P. (2012). Two-level Configuration for FPGA : A New Design Methodology Based on a Computing Fabric. *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*. 265 –268.

ALTERA (2007). Stratix III FPGAs vs. Xilinx Virtex-5 Devices : Architecture and Performance Comparison. White paper.

ANSALONI, G., BONZINI, P. et POZZI, L. (2011). EGRA : A Coarse Grained Reconfigurable Architectural Template. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19, 1062 –1074.

BERGERON, E., SAINT-MLEUX, X., FEELEY, M. et DAVID, J. (2005). High Level Synthesis for Data-Driven Applications. *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*. IEEE Computer Society, Washington, DC, USA, RSP '05, 54–60.

BJERREGAARD, T. et MAHADEVAN, S. (2006). A Survey of Research and Practices of Network-on-Chip. *ACM Comput. Surv.*, 38.

BONONI, L. et CONCER, N. (2006). Simulation and Analysis of Network on Chip Architectures : Ring, Spidergon and 2D Mesh. *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*. vol. 2, 6 pp.

CARMONA, J., CORTADELLA, J., KISHINEVSKY, M. et TAUBIN, A. (2009). Elastic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28, 1437–1455.

CHALAMALASETTI, S., PUROHIT, S., MARGALA, M. et VANDERBAUWHEDE, W. (2009). MORA - An Architecture and Programming Model for a Resource Efficient Coarse Grained Reconfigurable Processor. *2009 NASA/ESA Conference on Adaptive Hardware and Systems*, 389–396.

CHI, W., LAMOUREUX, J., WILTON, S., LEONG, P. et LUK, W. (2008). The Coarse-Grained / Fine-Grained Logic Interface in FPGAs with Embedded Floating-Point Arithmetic Units. *Programmable Logic, 2008 4th Southern Conference on*. 63 –68.

COMPTON, K. et HAUCK, S. (2002). Reconfigurable Computing : A Survey of Systems and Software. *ACM Computing Surveys*, 34, 171–210.

CRONQUIST, D., FRANKLIN, P., BERG, S. et EBELING, C. (1998). Specifying and Compiling Applications for RaPiD. *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*. 116 –125.

DENNIS, J. (1980). Data Flow Supercomputers. *Computer*, 13, 48–56.

DENNIS, J. et MISUNAS, D. (1975). A Preliminary Architecture for a Basic Data-flow Processor. *Proceedings of the 2nd annual symposium on Computer architecture*. ACM, New York, NY, USA, ISCA '75, 126–132.

DEVAUX, L., CHILLET, D., PILLEMENT, S. et DEMIGNY, D. (2009). Flexible Communication Support for Dynamically Reconfigurable FPGAS. *Programmable Logic, 2009. SPL. 5th Southern Conference on*. 65 –70.

EBELING, C., CRONQUIST, D. et FRANKLIN, P. (1996). RaPiD - Reconfigurable Pipelined Datapath.

GIEFERS, H. et PLATZNER, M. (2007). A Many-Core Implementation Based on the Reconfigurable Mesh Model. *2007 International Conference on Field Programmable Logic and Applications*, 41–46.

GOLDSTEI, S., SCHMIT, H., MOE, M., BUDIU, M., CADAMBI, S., TAYLOR, R. et LAUFER, R. (1999). PipeRench : A Coprocessor for Streaming Multimedia Acceleration. *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*. 28 –39.

GOLDSTEIN, S., SCHMIT, H., BUDI, M., CADAMBI, S., MOE, M. et TAYLOR, R. (2000). PipeRench : A Reconfigurable Architecture and Compiler. *Computer*, 33, 70 –77.

HARTENSTEIN, R. (2001). Coarse Grain Reconfigurable Architecture (embedded tutorial). *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. ACM, New York, NY, USA, ASP-DAC '01, 564–570.

HOSSEINABADY, M. et NUNEZ-YANEZ, J. (2008). Fault-tolerant Dynamically Reconfigurable NoC-based SoC. *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*. 31 –36.

JOHNSTON, W., HANNA, J. et MILLAR, R. (2004). Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36, 1–34.

KILLIAN, C., TANOUGAST, C., MONTEIRO, F. et DANDACHE, A. (2012). A New Efficient and Reliable Dynamically Reconfigurable Network-on-chip. *Journal of Electrical and Computer Engineering*, 2012.

KINSY, M., SUH, G. et DEVADAS, S. (2008). Diastolic Arrays : Throughput-driven Reconfigurable Computing. *2008 IEEE/ACM International Conference on Computer-Aided Design*, 457–464.

LANUZZA, M., PERRI, S. et CORSONELLO, P. (2007). A New Reconfigurable Coarse-Grain Architecture for Multimedia Applications. *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, 119–126.

PARHAMI, B. (1999). *Introduction to Parallel Processing : Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA.

PARK, H., PARK, Y. et MAHLKE, S. (2009). A Dataflow-Centric Approach to Design Low Power Control Paths in CGRAs. *2009 IEEE 7th Symposium on Application Specific Processors*, 15–20.

PARVEZ, H., MARRAKCHI, Z. et MEHREZ, H. (2008). Enhanced Methodology and Tools for Exploring Domain-Specific Coarse-Grained FPGAs. *2008 International Conference on Reconfigurable Computing and FPGAs*, 121–126.

SANEEI, M., AFZALI-KUSHA, A. et NAVABI, Z. (2006). Low-latency Multi-Level Mesh Topology for NoCs. *Microelectronics, 2006. ICM '06. International Conference on*. 36 –39.

SINGH, H., MING-HAU, L., GUANGMING, L., KURDAHI, F., BAGHERZADEH, N. et FILHO, E. C. (2000). MorphoSys : An Integrated Reconfigurable System for Data-parallel and Computation-intensive Applications. *Computers, IEEE Transactions on*, 49, 465 –481.

STERLING, T., KUEHN, J., THISTLE, M. et ANASTASIS, T. (1995). *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press.

STRANNEBY, D. et WALKER, W. (2004). *Digital Signal Processing And Applications*. ELSEVIER.

SVENSSON, B. (2009). Evolution in Architectures and Programming Methodologies of Coarse-Grained Reconfigurable Computing. *Microprocessors and Microsystems*, 33, 161–178.

SYLVESTER, D. et KEUTZER, K. (2000). A Global Wiring Paradigm for Deep Submicron Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19, 242–252.

THIELMANN, B., HUTHMANN, J. et KOCH, A. (2011). Precore - A Token-Based Speculation Architecture for High-Level Language to Hardware Compilation. *2011 21st International Conference on Field Programmable Logic and Applications*, 123–129.

TODMAN, T., CONSTANTINIDES, G., WILTON, S., MENCER, O., LUK, W. et CHEUNG, P. (2005). Reconfigurable Computing : Architectures and Design Methods. *Computers and*, 152, 193–207.

WAINGOLD, E., TAYLOR, M., SRIKRISHNA, D., SARKAR, V., LEE, W., LEE, V., KIM, J., FRANK, M., FINCH, P., BARUA, R., BABB, J., AMARASINGHE, S. et AGARWAL, A. (1997). Baring it all to Software : Raw Machines. *Computer*, 30, 86 –93.

WATSON, I. (1979). A Prototype Data Flow Computer with Token Labelling. *afips*, 623.

## PUBLICATIONS DE L'AUTEUR

ALLARD, M., GROGAN, P., DAVID, J.-P. (2009). A Scalable Architecture for Multivariate Polynomial Evaluation on FPGA. *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on.* 107 - 112.

ALLARD, M., GROGAN, P., SAVARIA, Y. et DAVID, J.-P. (2012). Two-level Configuration for FPGA : A New Design Methodology Based on a Computing Fabric. *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on.* 265 - 268.